MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

25 Binary Search Trees

Department of Computing and Software

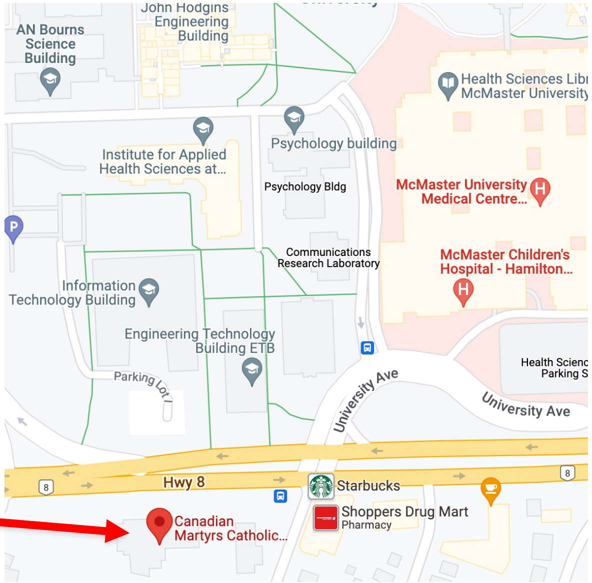
Instructor:

Omid Isfahanialamdari

March 21, 2022

Admin

- Mid-Term 2:
  - Wednesday 23 March 2022
  - Duration: **1 hour**
  - **From 1:30 to 14:30 (lec. time)**
  - Location: **MCMST CDN_MARTYRS**
    - Seems to be here, I am not sure

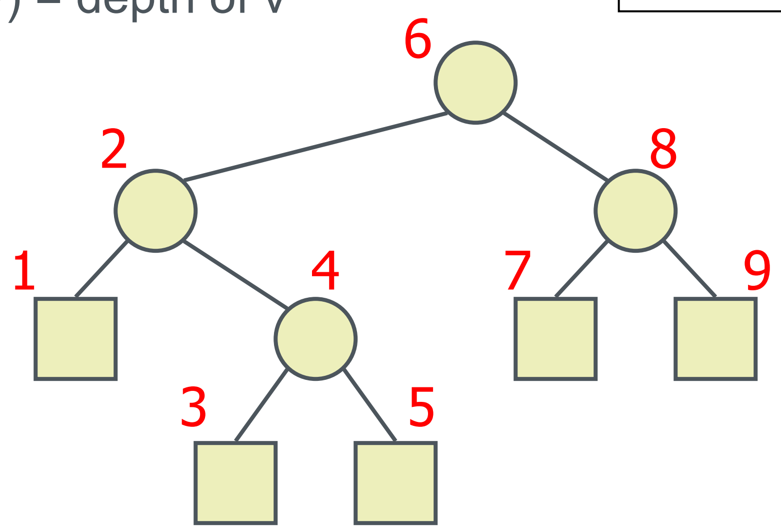


  - Covers: Topics from “Doubly Linked Lists” until the lecture of Wednesday 16 March 2022 (inclusive)

Inorder Traversal - From lec. on Binary Trees

- In an inorder traversal, a node is visited after its left subtree and before its right subtree.
- Application: draw a binary tree with the following **coordinates**:
    - x(v) = inorder rank of v
    - y(v) = depth of v

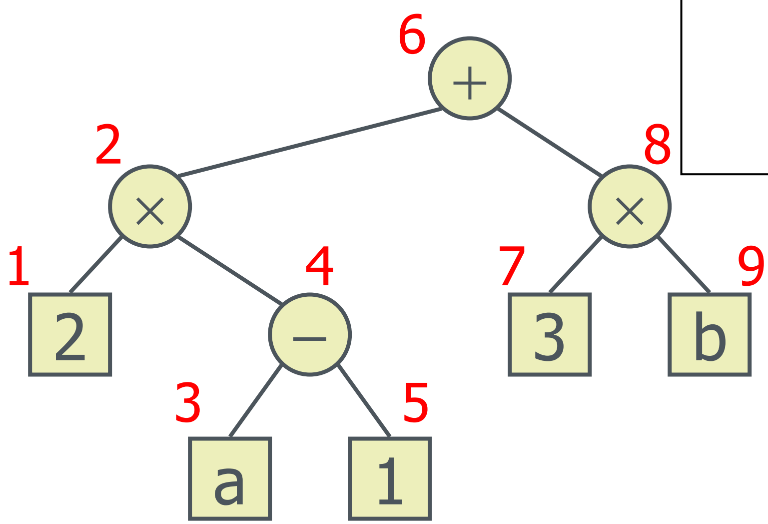
Algorithm *inOrder*(*v*)
    if ¬ *v.isExternal*()
        inOrder(*v.left*())
    visit(*v*)
    if ¬ *v.isExternal*()
        inOrder(*v.right*())



Print Arithmetic Expressions - Binary Trees

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print “(“ before traversing left subtree
  - print “)“ after traversing right subtree

Algorithm *printExpression(v)*
    if $\neg$*v.isExternal*()
        print(“(”)
        printExpression(*v.left*())
    print(*v.element*())
    if $\neg$*v.isExternal*()
        printExpression(*v.right*())
        print (“)”)

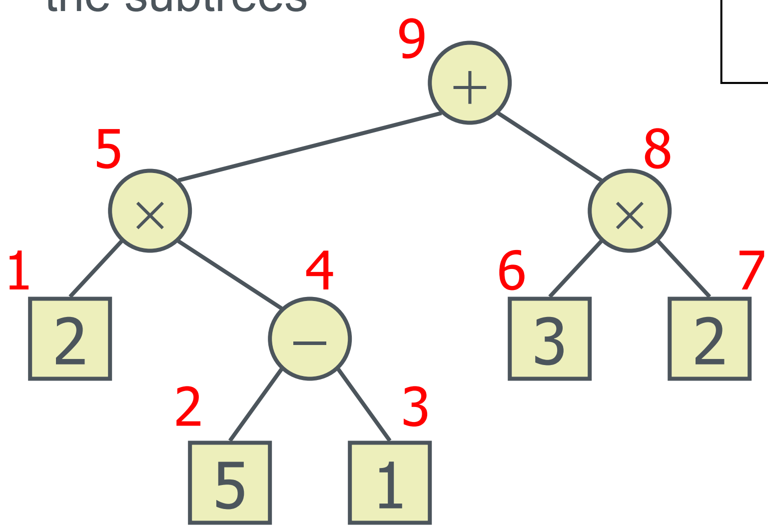


$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions - Binary Trees

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

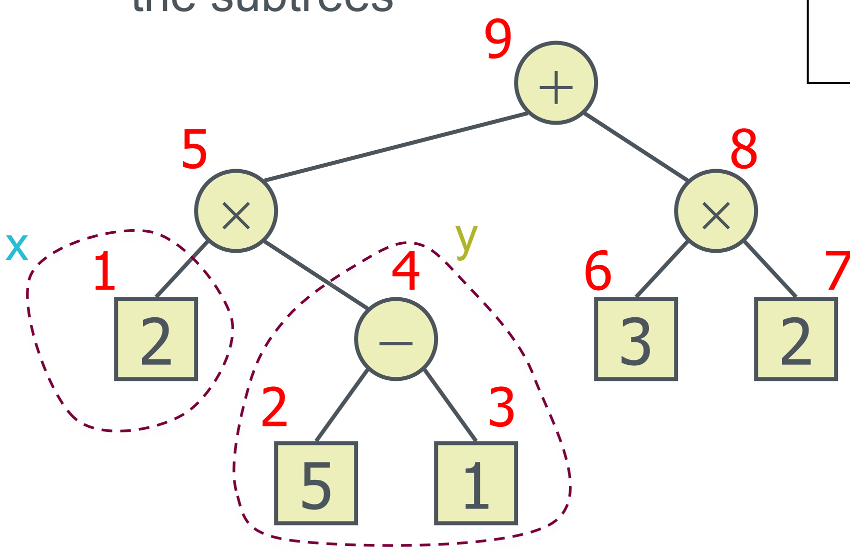
Algorithm *evalExpr*(*v*)
    if *v.isExternal*()
        return *v.element*()
    else
        $x \leftarrow$ *evalExpr*(*v.left*())
        $y \leftarrow$ *evalExpr*(*v.right*())
        $\Diamond \leftarrow$ operator stored at v
        return $x \Diamond y$



Evaluate Arithmetic Expressions - Binary Trees

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

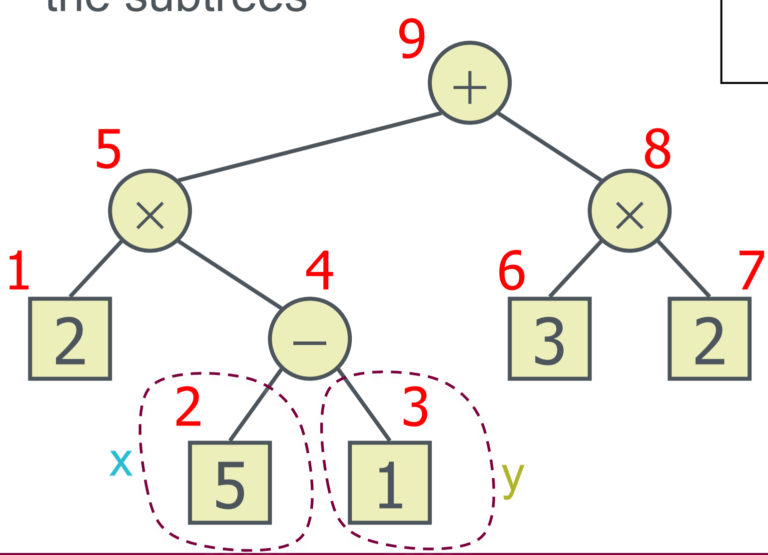
Algorithm *evalExpr*(*v*)
    if *v.isExternal*()
        return *v.element*()
    else
        $x \leftarrow$ *evalExpr*(*v.left*())
        $y \leftarrow$ *evalExpr*(*v.right*())
        $\Diamond \leftarrow$ operator stored at v
        return $x \Diamond y$

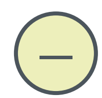


Evaluate Arithmetic Expressions - Binary Trees

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(*v*)
    if *v.isExternal*()
        return *v.element*()
    else
        x ← *evalExpr*(*v.left*())
        y ← *evalExpr*(*v.right*())
        ◊ ← operator stored at *v*
        return *x* ◊ *y*



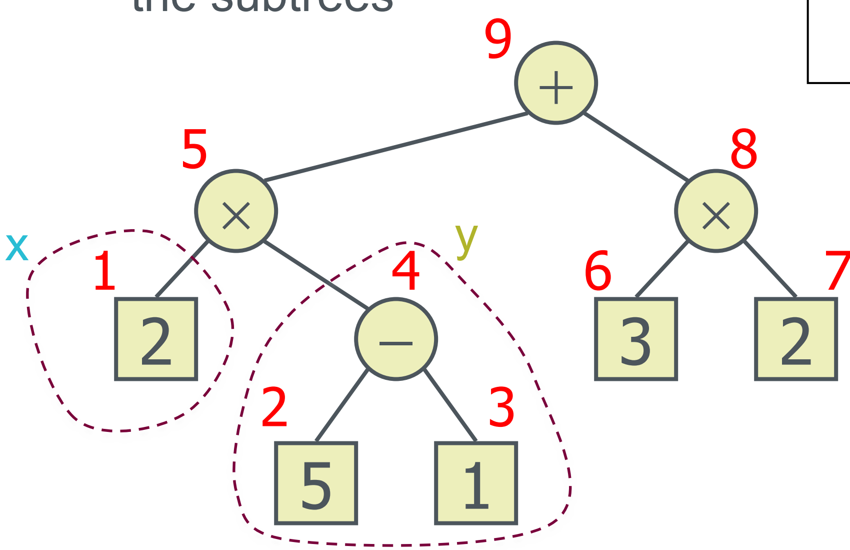
v is 

x ← 5
y ← 1
return 5 - 1

Evaluate Arithmetic Expressions - Binary Trees

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
  if $v.isExternal()$
    return $v.element()$
  else
    $x \leftarrow$ *evalExpr*($v.left()$)
    $y \leftarrow$ *evalExpr*($v.right()$)
    $\Diamond \leftarrow$ operator stored at v
    return $x \Diamond y$



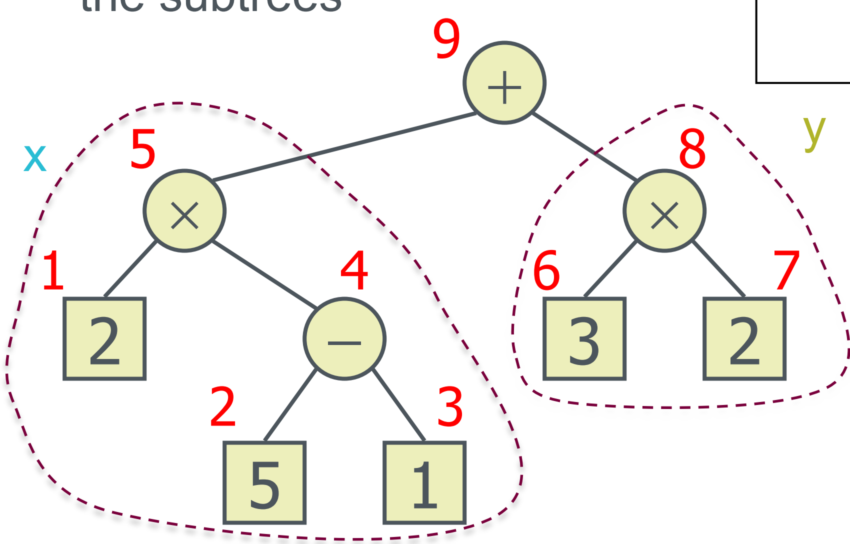
v is ×

$x \leftarrow 2$
$y \leftarrow 4$
return 2×4

Evaluate Arithmetic Expressions - Binary Trees

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

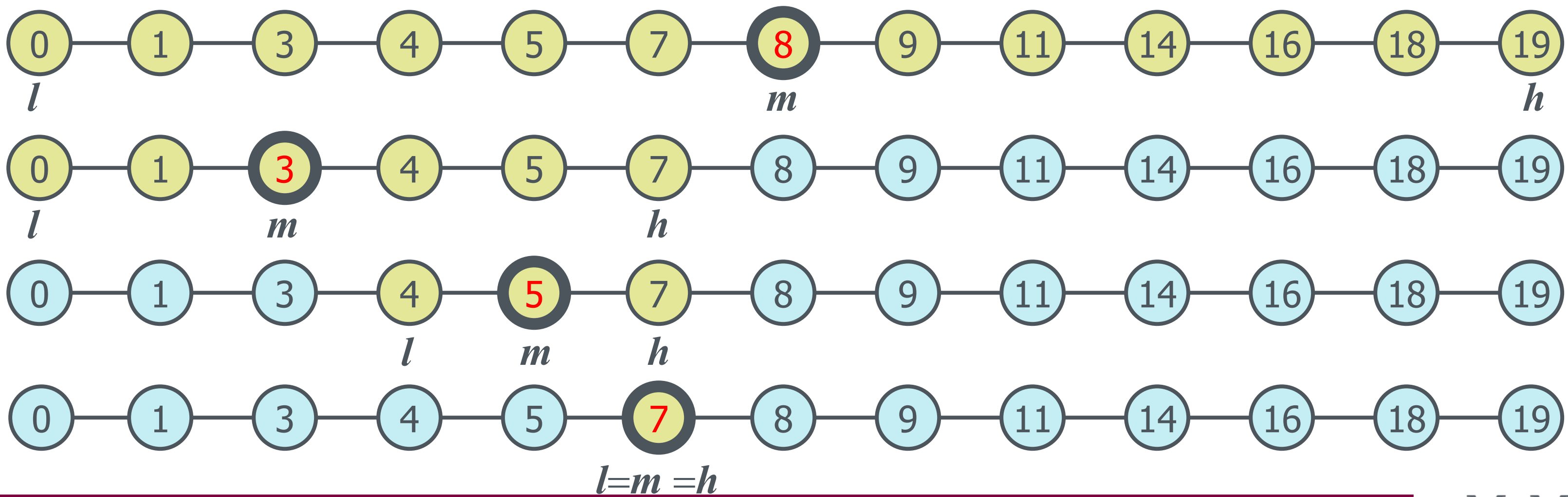
Algorithm *evalExpr*(*v*)
    if *v.isExternal*()
        return *v.element*()
    else
        x ← *evalExpr*(*v.left*())
        y ← *evalExpr*(*v.right*())
        ◊ ← operator stored at *v*
        return *x* ◊ *y*



- subtree x evaluates to 8
- subtree y evaluates to 6
- the whole tree evaluates to 14

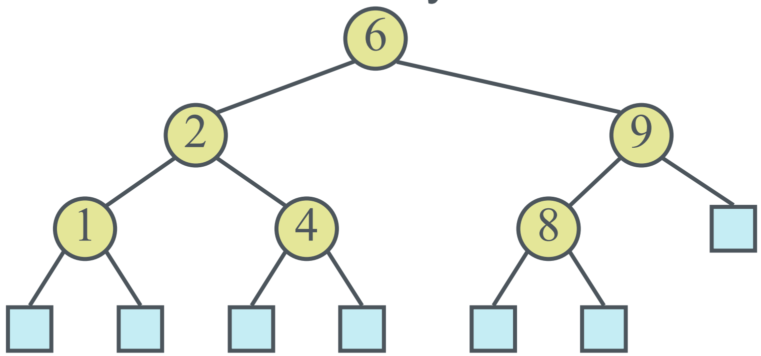
Binary Search - will be discussed later in more detail

- We can do a fast search on an array that is already sorted by keys
- It works by repeatedly halving the portion of the array that can contain the item, until we narrowed down the portion to have just one element.
  - At each step, the number of candidate items is halved
  - terminates after O(log n) steps
  - Example: find(7)



Binary Search Tree

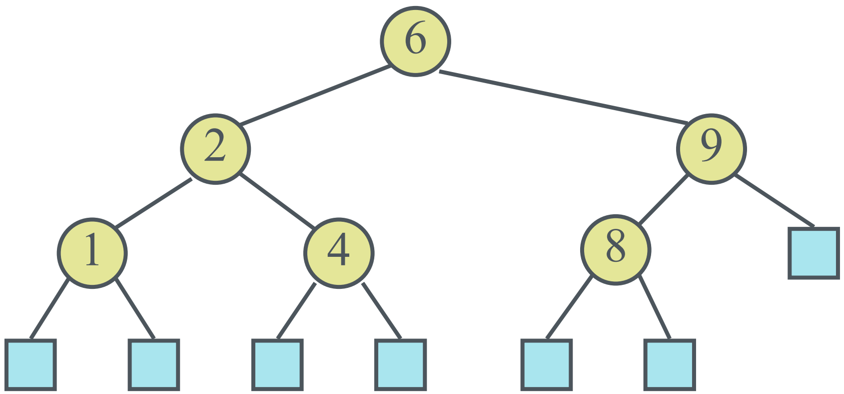
- A **binary search tree** is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let **u**, **v**, and **w** be three nodes such that **u** is in the left subtree of **v** and **w** is in the right subtree of **v**. We have:
    - key(u) ≤ key(v) ≤ key(w)
  - External nodes do not store items
    - This approach simplifies several of our search and update algorithms
- An inorder traversal of a binary search trees visits the keys in **non-decreasing** order.



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(4):
  - Call TreeSearch(4,root)

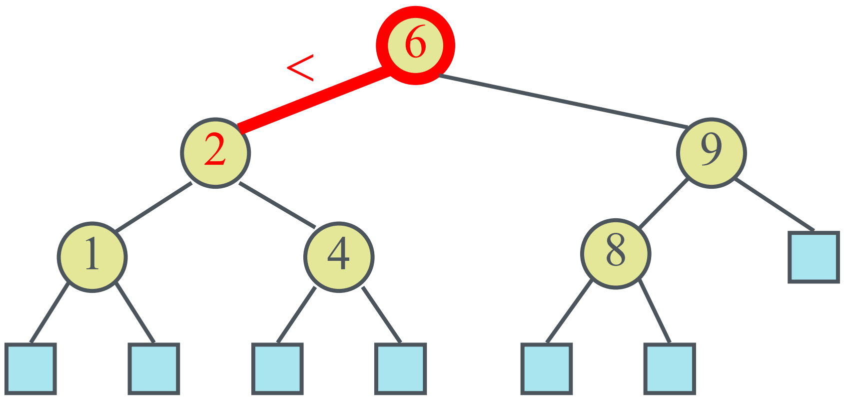
Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch(k, T.left(v))
    else if $k >$ key(v) **then**
        return TreeSearch(k, T.right(v))
    return v {we know $k =$ key(v)}



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(4):
    - Call TreeSearch(4,root)

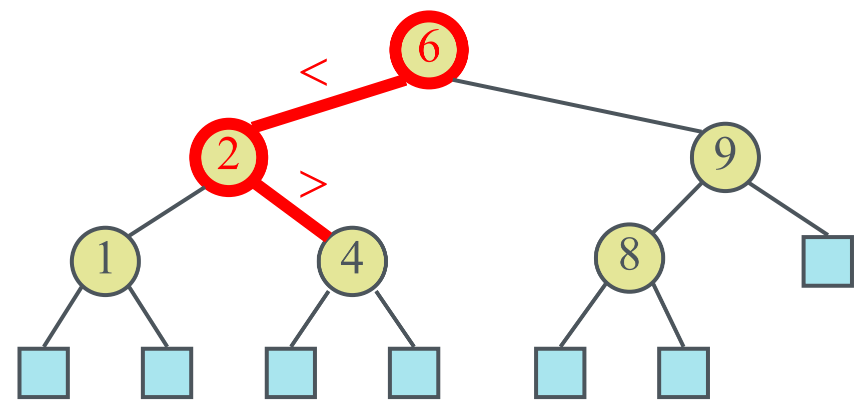
Algorithm TreeSearch(k, v):
  if T.isExternal(v) **then**
    return v
  if $k <$ key(v) **then**
    return TreeSearch$(k, T.\text{left}(v))$
  else if $k >$ key(v) **then**
    return TreeSearch$(k, T.\text{right}(v))$
  return v $\{$we know $k = \text{key}(v)\}$



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(4):
  - Call TreeSearch(4,root)

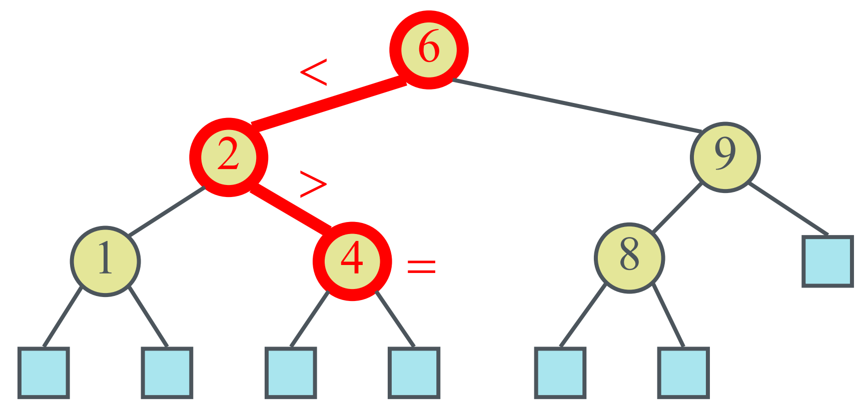
Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T.\text{left}(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T.\text{right}(v))$
    return v $\{$we know $k =$ key$(v)\}$



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(4):
  - Call TreeSearch(4,root)

Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T.\text{left}(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T.\text{right}(v))$
    return v {we know $k =$ key(v)}

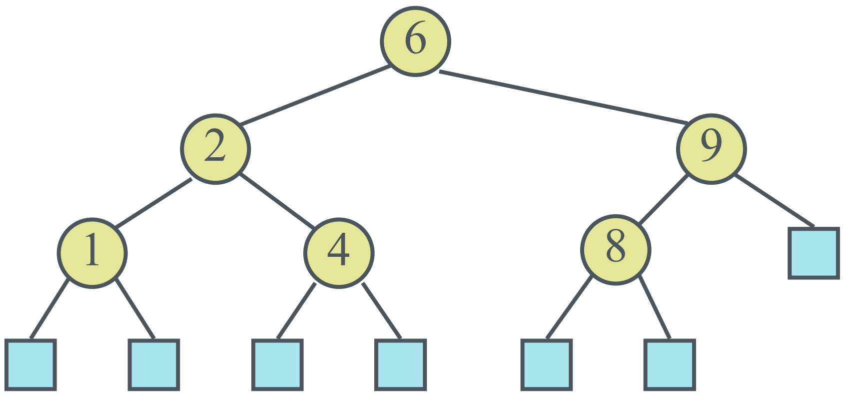


McMaster University

Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(5):
  - Call TreeSearch(5,root)

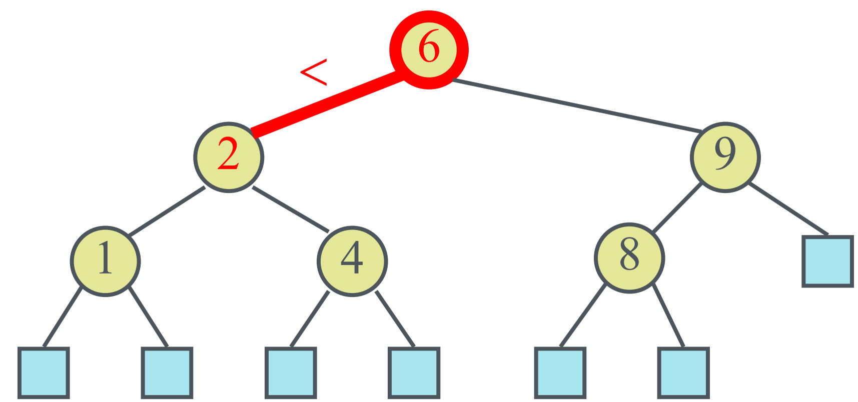
Algorithm TreeSearch(k, v):
  if T.isExternal(v) **then**
    return v
  if $k <$ key(v) **then**
    return TreeSearch$(k, T$.left$(v))$
  else if $k >$ key(v) **then**
    return TreeSearch$(k, T$.right$(v))$
  return v {we know $k =$ key(v)}



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(5):
  - Call TreeSearch(5,root)

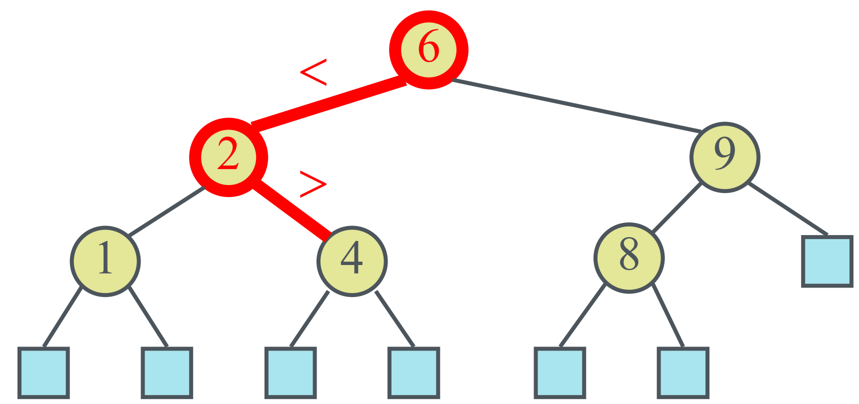
Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T.\text{left}(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T.\text{right}(v))$
    return v $\{$we know $k =$ key$(v)\}$



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(5):
  - Call TreeSearch(5,root)

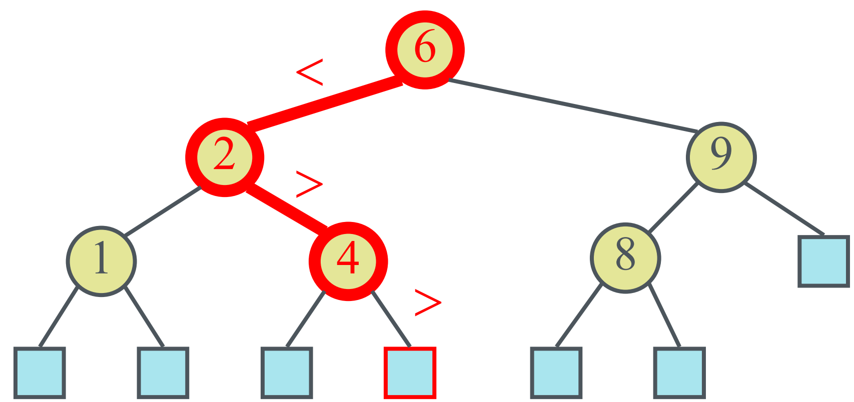
Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T$.left$(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T$.right$(v))$
    return v {we know $k =$ key(v)}



Binary Search Tree - Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of **k** with the key of the current node
- If we reach a leaf, the key is not found (unsuccessful)
- Example: get(5):
  - Call TreeSearch(5,root)
    - unsuccessful!

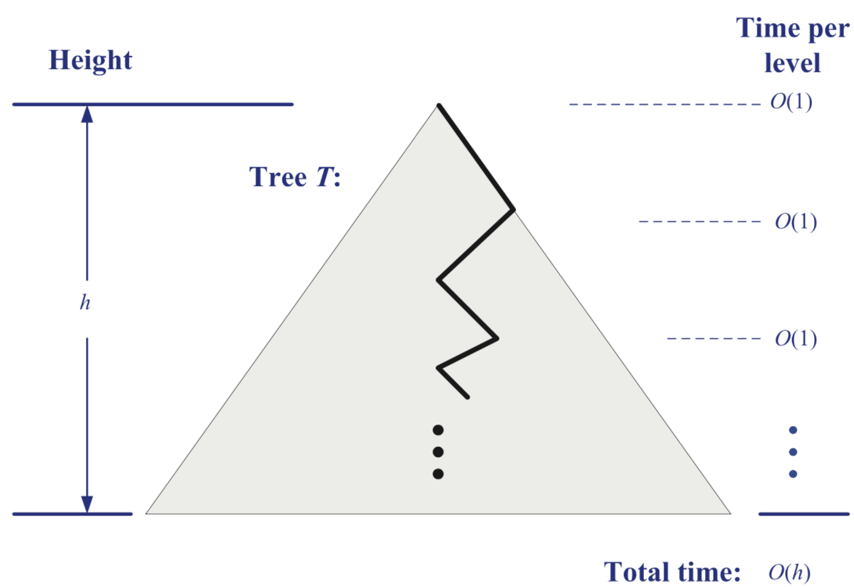
Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T.\text{left}(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T.\text{right}(v))$
    return v {we know $k =$ key(v)}



Binary Search Tree - Analysis of Search

- executes a constant number of primitive operations for each recursive call
- That is, TreeSearch is called on the nodes of a path of *T* that starts at the root and goes down one level at a time => **O(h)**

Algorithm TreeSearch(k, v):
  if T.isExternal(v) **then**
    return v
  if $k <$ key(v) **then**
    return TreeSearch$(k, T$.left$(v))$
  else if $k >$ key(v) **then**
    return TreeSearch$(k, T$.right$(v))$
  return v {we know $k =$ key(v)}

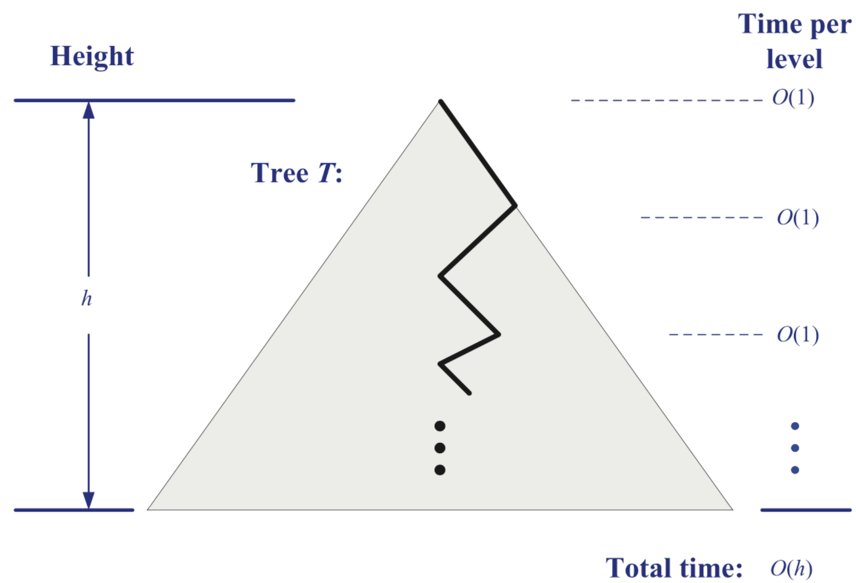


Binary Search Tree - Analysis of Search

- **executes a constant number of primitive operations for each recursive call**
- **That is, TreeSearch is called on the nodes of a path of *T* that starts at the root and goes down one level at a time => O(h)**

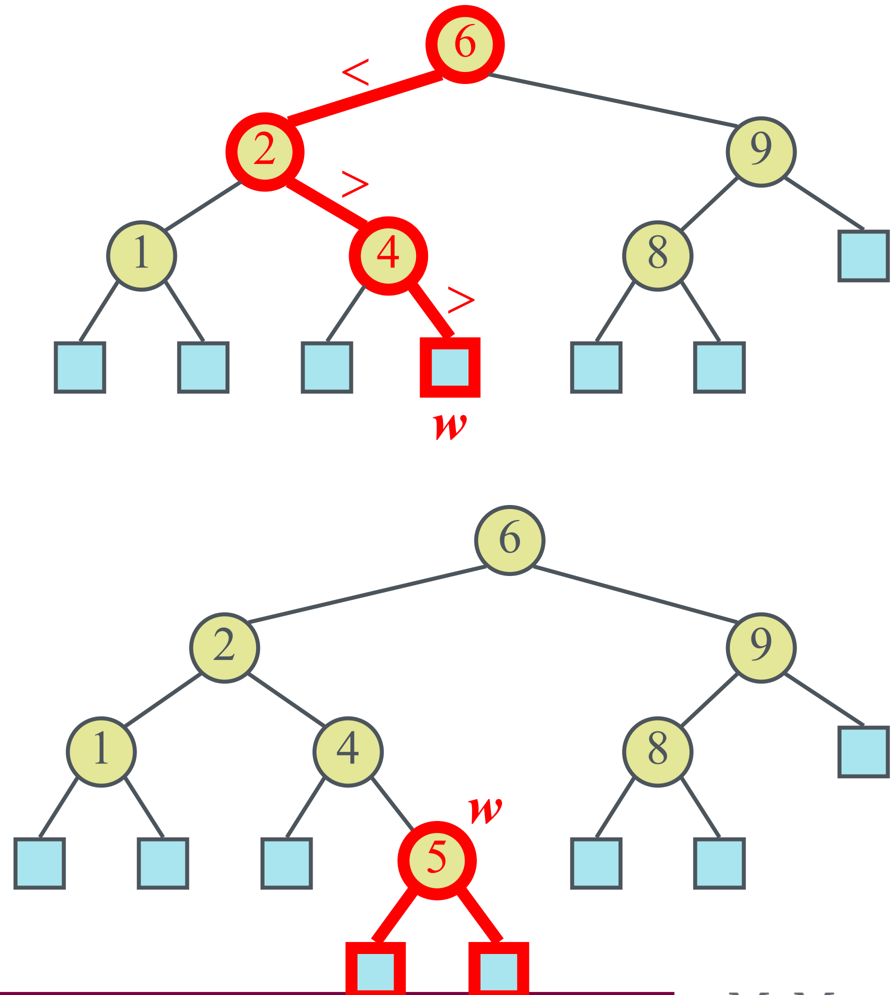
Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T.$left$(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T.$right$(v))$
    return v {we know $k =$ key(v)}

- Recall that the height of a tree with ***n*** nodes can be as small as ***O*(log*n*)** or as large as **O(*n*)**
  - binary search trees are most efficient when they have small height.



Binary Search Tree - Insert

- To perform operation **put(k, o)**, we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



Questions?