Week 4
Stacks, Queues, and Trees

Algorithms and Data Structures

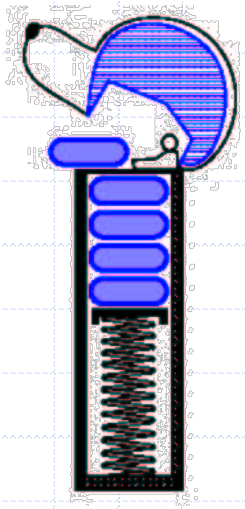
COMP3506/7505

Week 4 – Stacks, Queues, Trees

1. Stacks
2. Queues
3. Tree basics
4. Tree traversals
5. Binary trees

Stack ADT

- Stores arbitrary objects
- Insertions and deletions are last-in, first-out (LIFO)
- Think of a spring-loaded plate dispenser
  - insertions at top of stack
  - removals from top of stack



Stack Operations (ADT)

- `push(T)`
- `T pop()`
- `T top()`
- `integer size()`
- `boolean isEmpty()`

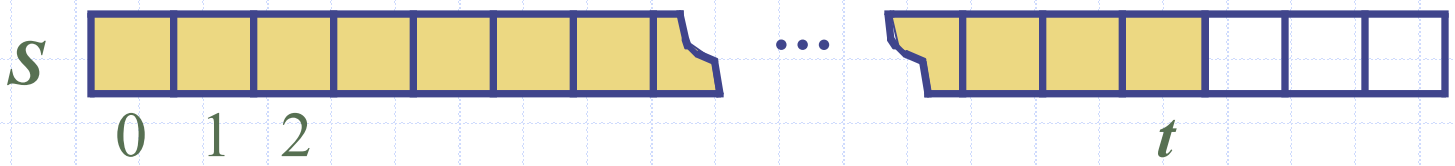
Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Array-based Stack

- Simple Stack implementation
- Add elements from left to right
- Variable keeps track of index of top element

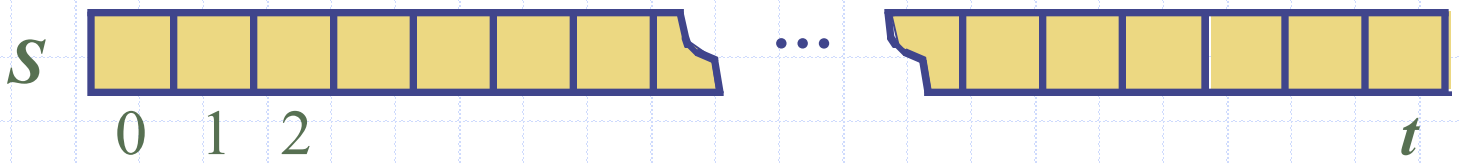
```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        throw
            NoSuchElementException
    else
        t ← t – 1
        return S[t + 1]
```



Array-based Stack (cont.)

- Array storing the stack may become full
- Push operation throws IllegalStateException
  - limitation of array implementation
  - not intrinsic to the Stack ADT
  - could specialise to be StackOverflowException

```
Algorithm push(item)
    if t = S.length – 1 then
        throw
            IllegalStateException
    else
        t ← t + 1
        S[t] ← item
```



Performance

- Let n be the number of elements in the stack, and let $N \geq n$ be the fixed size of the array
- Space used is O(N)
- Each operation runs in time O(1)
  - Push to the back is O(1) – We know where the back is, we just look up that index and put the element there
  - Pop is O(1) – We know where the back is, we just look up that index and return the element there
  - Empty/Full/Size are simple checks
- Typically, the size (the number of elements on the stack) is maintained inside the structure

Other Implementations

- We could use an extensible list
  - Amortized O(1) push – why?
  - O(1) pop
- We could use some sort of linked list
  - O(1) push and pop

Week 4 – Stacks, Queues, Trees

1. Stacks
2. Queues
3. Tree basics
4. Tree traversals
5. Binary trees

Queue ADT

- Stores arbitrary objects
- Insertions and deletions are first-in first-out (FIFO)
- Think of a waiting line (queue)
  - insertions at rear of queue
  - removals at front of queue

Queue Operations

- enqueue(T)
- T dequeue()

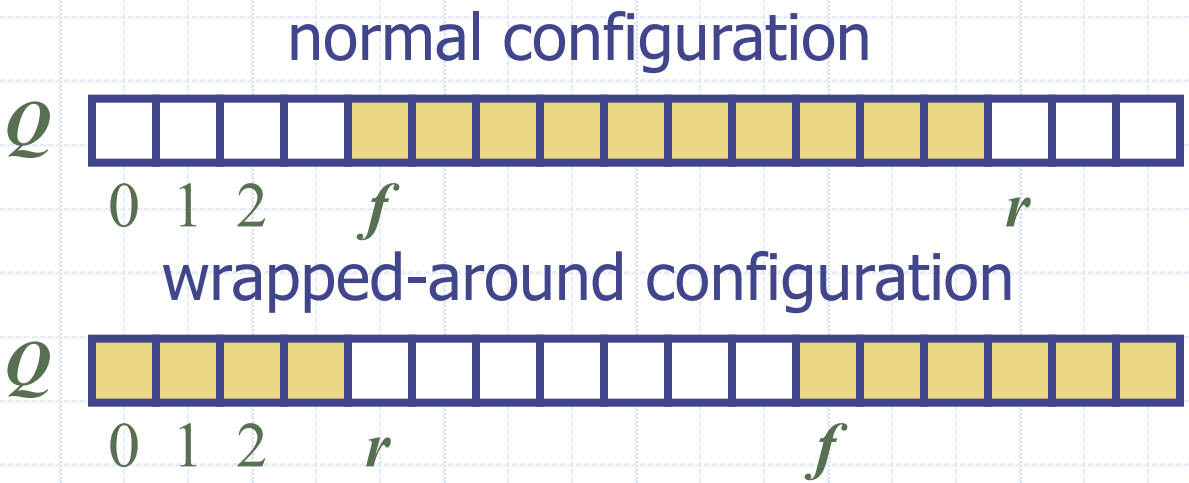
- T first()
- integer size()
- boolean isEmpty()

Example

Operation	*Return*	*Queue Contents*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Array-based Queue

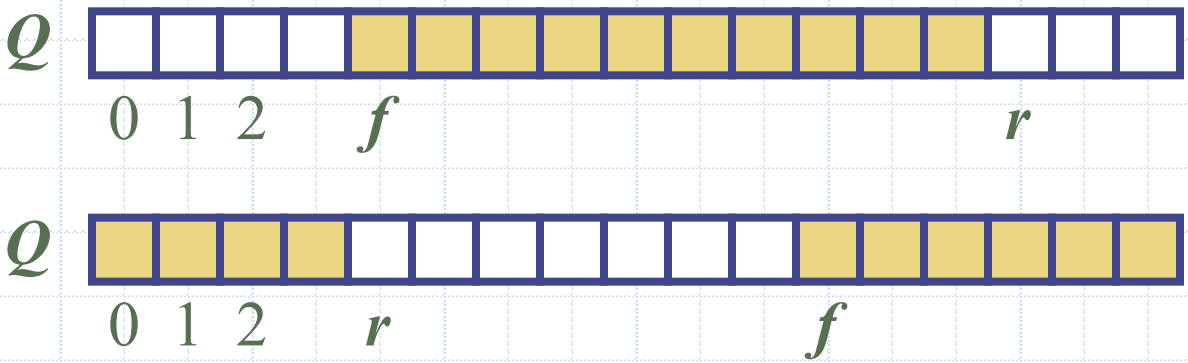
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
  - $r = (f + sz) \bmod N$
  - ***N=10, f=1, sz=8***
  - ***N=10, f=5, sz=8***



Queue Operations

Algorithm ***size***()
  return *sz*

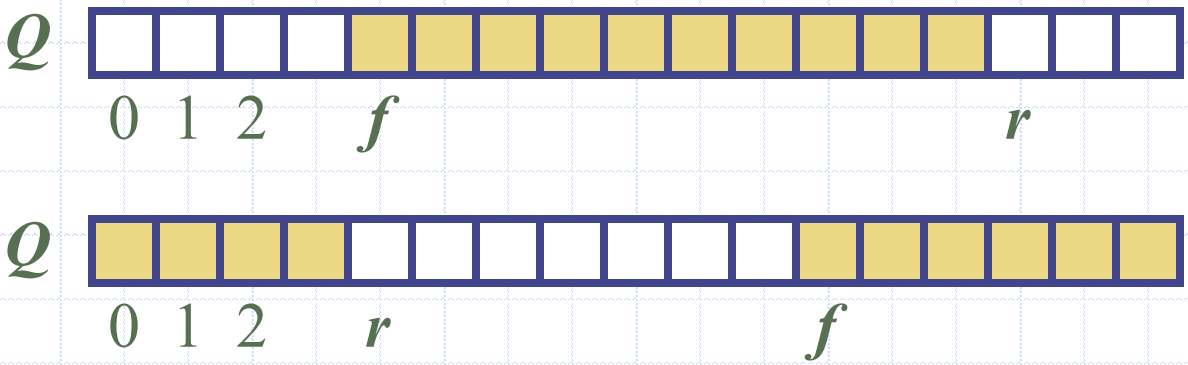
Algorithm ***isEmpty***()
  return (*sz* == 0)



Queue Operations (cont.)

- enqueue throws an exception if array is full
  - implementation-dependent
- Uses modulo operator
  - remainder of division

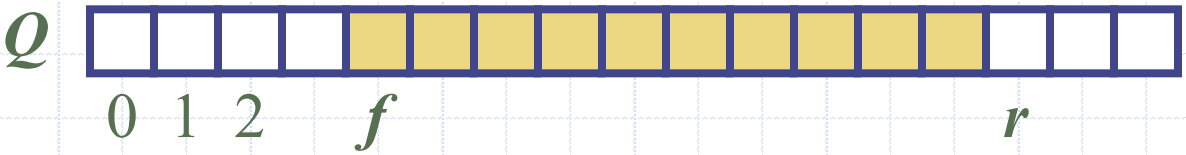
Algorithm *enqueue*(*item*)
  if *size*() = *N* **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz)$ **mod** N
    $Q[r] \leftarrow item$
    $sz \leftarrow (sz + 1)$



Queue Operations (cont.)

- Note that dequeue returns null if the queue is empty

Algorithm *dequeue*()
  if *isEmpty*() **then**
    return *null*
  else
    item ← *Q*[*f*]
    f ← (*f* + 1) **mod** *N*
    sz ← (*sz* – 1)
    return *item*

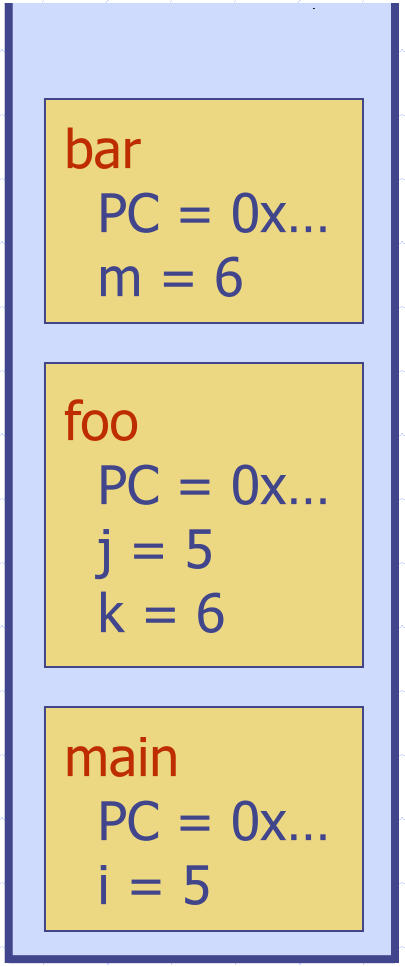


Some Stack and Queue Applications?

Programming: Call Stacks

- Programming languages use stacks to keep track of the current execution
- As methods are called, they are pushed to the call stack including information about the memory address of the instruction being executed (or the method being called) and the local variables required

```
main():
   i = 5
   foo(i)

foo(j):
   k = j+1
   bar(k)

bar(m):
   …
```



Reversing a List

```
def reverse(inlist):
  stack = []
  for item in inlist:
    stack.append(item)
  i = 0
  while len(stack) > 0:
    inlist[i] = stack.pop()
    i += 1
  return inlist


reverse([1,2,3,4,5,6])
[6, 5, 4, 3, 2, 1]
```

Note how the built-in list type (aka []) allows us to automatically implement stack-like behaviour via append and pop

Reversing a List

Alternatively use ArrayDeque

```
public static List<Integer> reverse(List<Integer> inlist) {
    Stack<Integer> stack = new Stack<>();
    for (int item : inlist) {
        stack.push(item);
    }

    int i = 0;
    while (!stack.isEmpty()) {
        inlist.set(i, stack.pop());
        i++;
    }
    return inlist;
}

reverse(new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6)));
// [6, 5, 4, 3, 2, 1]
```

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parenthesis Matching

```
def matched(in_string):
    mapping = {"(":")", "{":"}", "[":"]"}   # Note the dict used here
    print (mapping["("])                    # prints ')'
    stack = []
    for char in in_string:
        if char in mapping:
            stack.append(mapping[char]) # append the opposite bracket
        else:
            if len(stack) == 0:
                return False
            if char != stack.pop():
                return False
    if len(stack) == 0:
        return True
    return False
```

Parenthesis Matching

```
public static boolean matched(String inString) {
    Deque<Character> stack = new ArrayDeque<>();

    for (char c : inString.toCharArray()) {
        if (c == '(') stack.push(')');
        else if (c == '{') stack.push('}');
        else if (c == '[') stack.push(']');
        else {
            if (stack.isEmpty()) return false;
            if (c != stack.pop()) return false;
        }
    }
    return stack.isEmpty();
}

matched("([)]");    // false
```

HTML Tag Matching

- Correct XHTML: each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

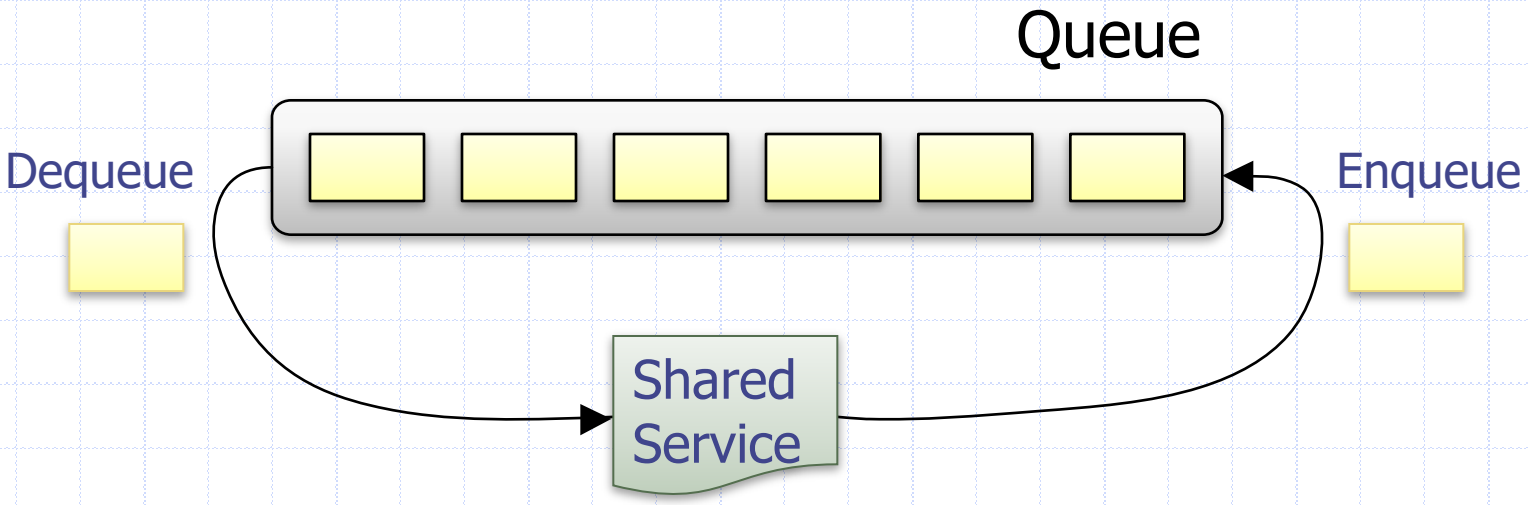
The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Applications of Queues: Round Robin Scheduler

- Can be implemented using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Later: *Priority Queues*

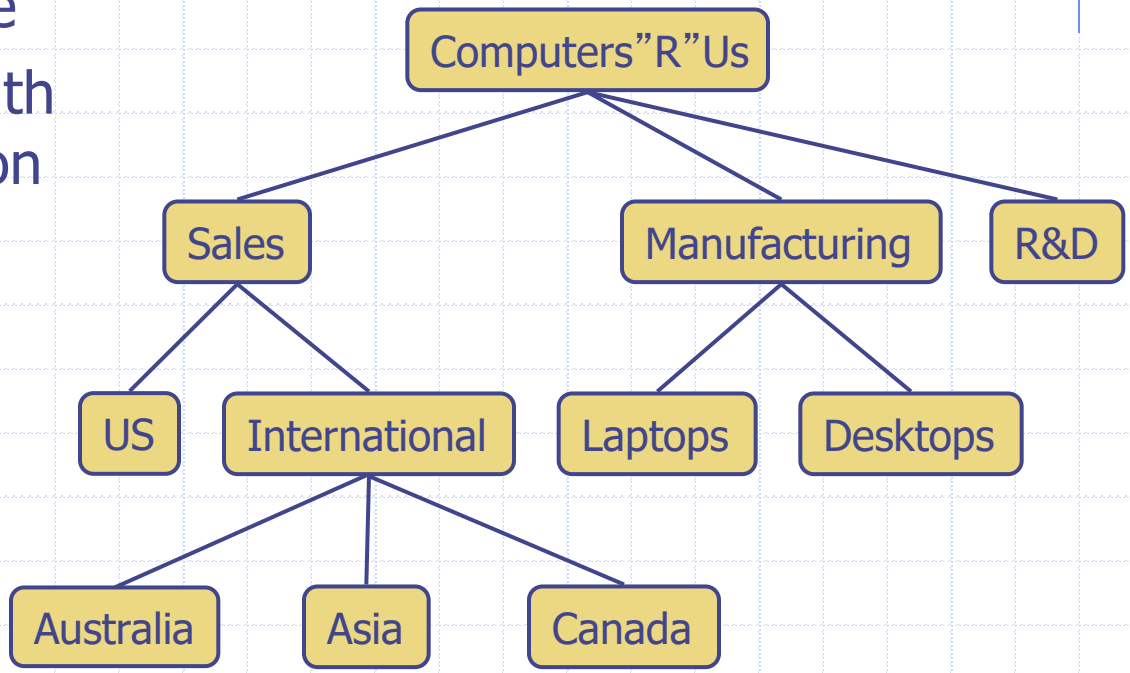
- One problem with the queue we are looking at here is that we can never change the priority – we always yield a FIFO ordering.
- Consider an emergency room – patients must be triaged according to how severe their affliction is. Does "cutting in line" feel appropriate here?
- Priority queues allow us to "cut in line" ☺

Week 4 – Stacks, Queues, Trees

1. Stacks
2. Queues
3. Tree basics
4. Tree traversals
5. Binary trees

What is a Tree

- Abstract model of a hierarchical structure
- Consists of nodes with a parent-child relation



Comparison Tree

Hierarchical Structure of Book

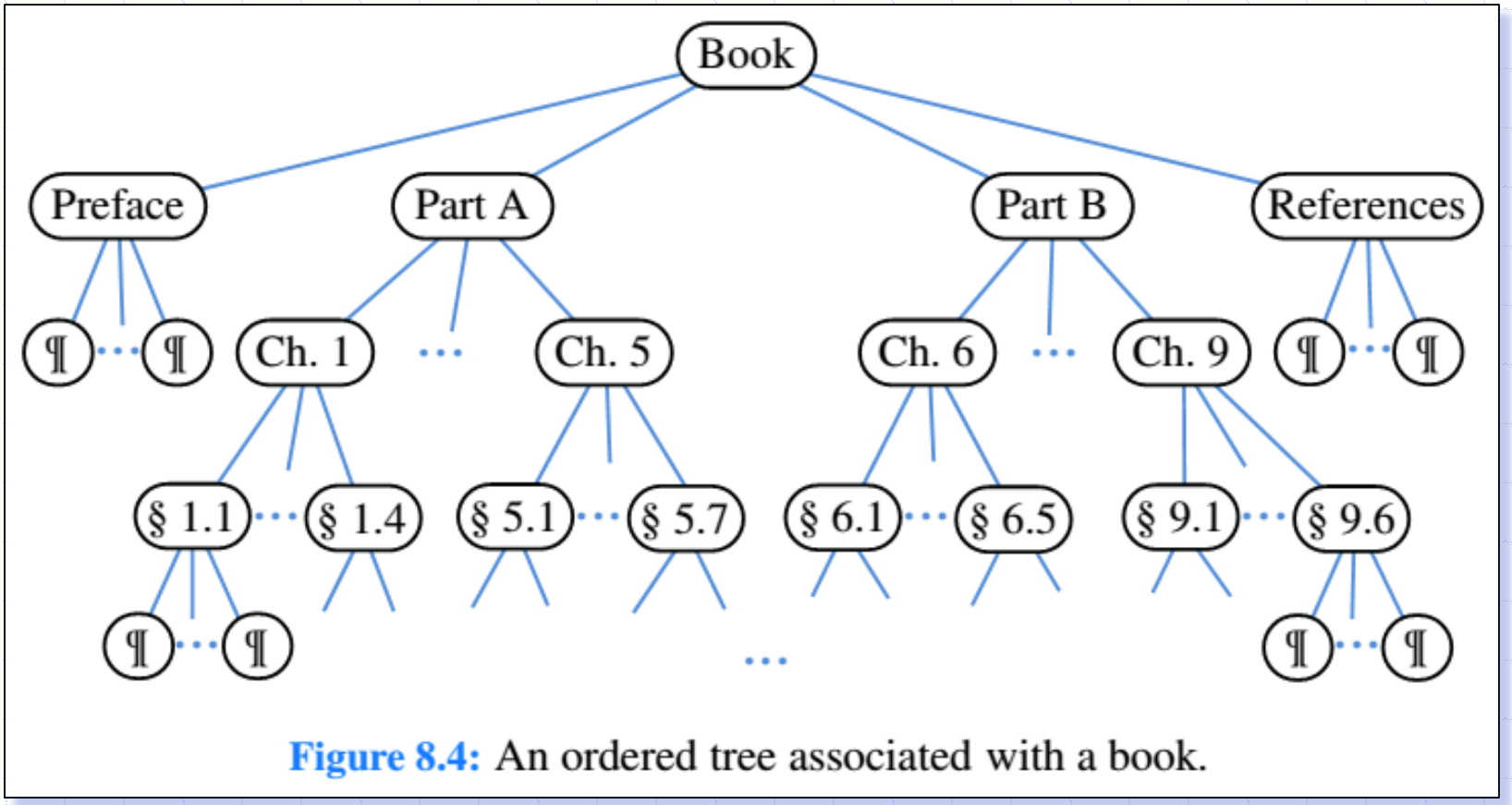
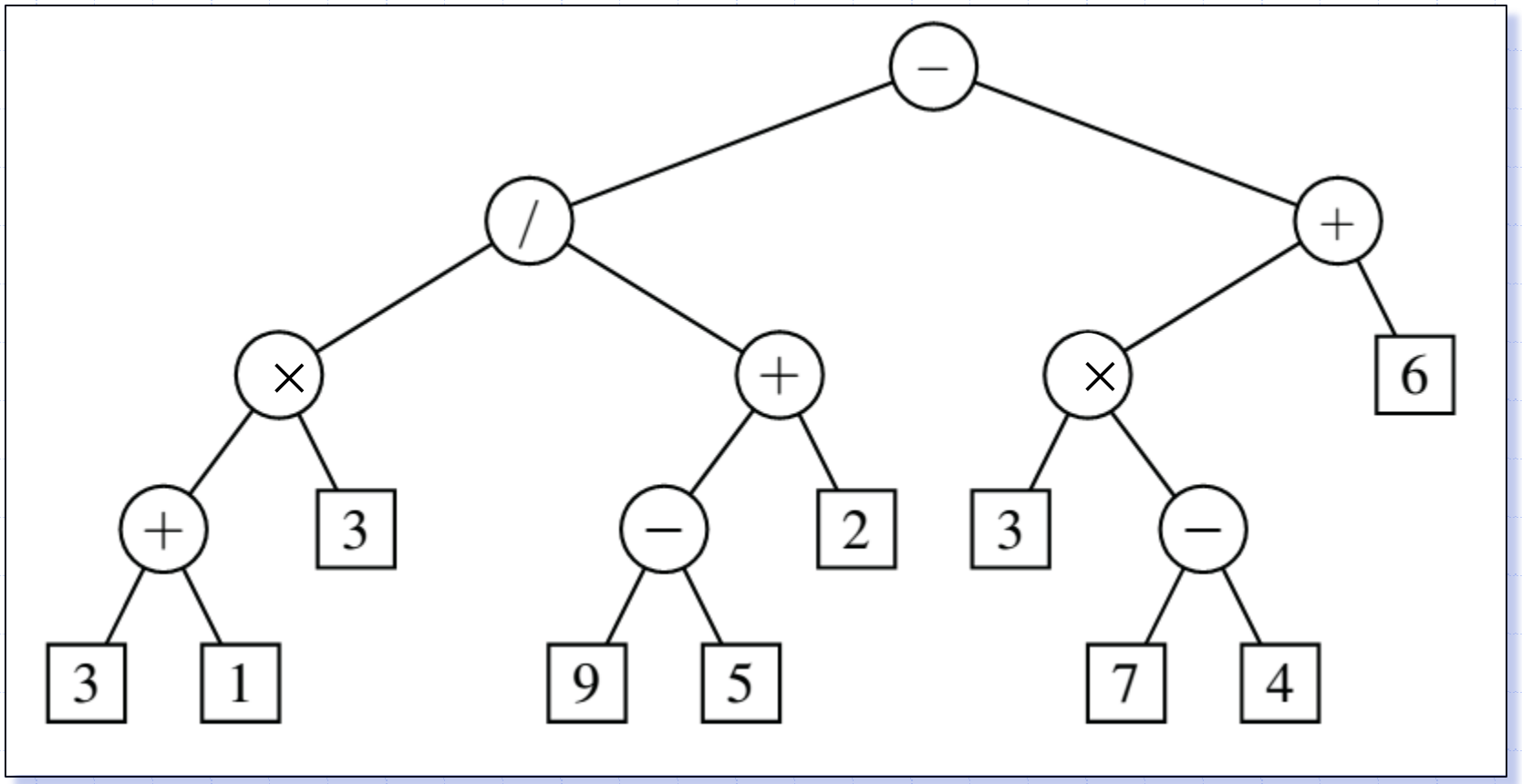


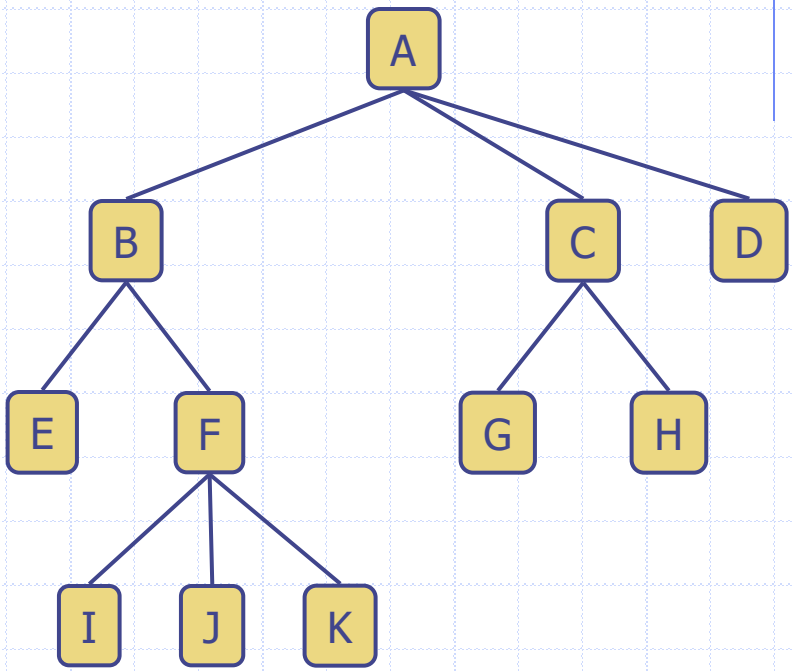
Figure 8.4: An ordered tree associated with a book.

Arithmetic Expression Tree



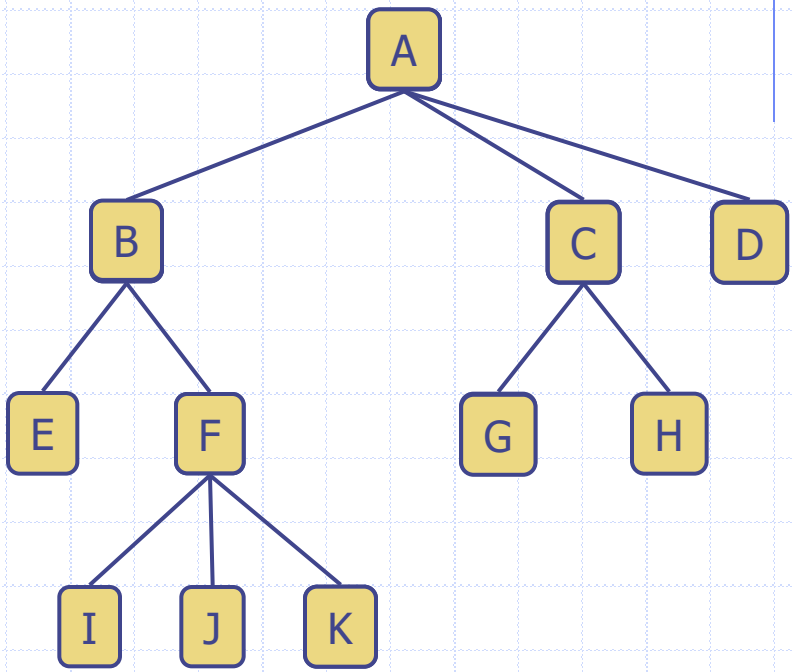
Tree Terminology

- Root
- Internal node
- External node (a.k.a. leaf)
- Edge



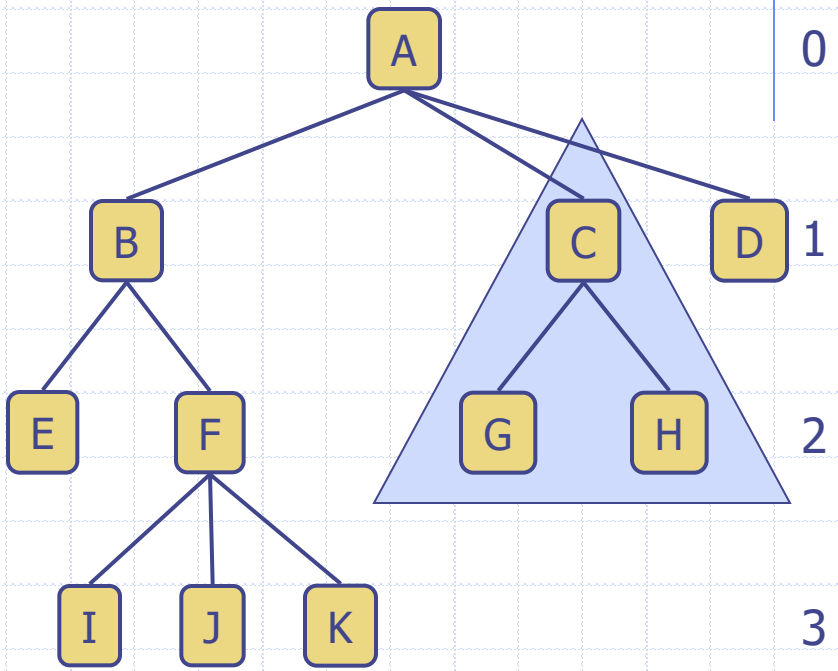
Tree Terminology

- Ancestors of a node
- Descendant of a node
- Siblings
- Degree



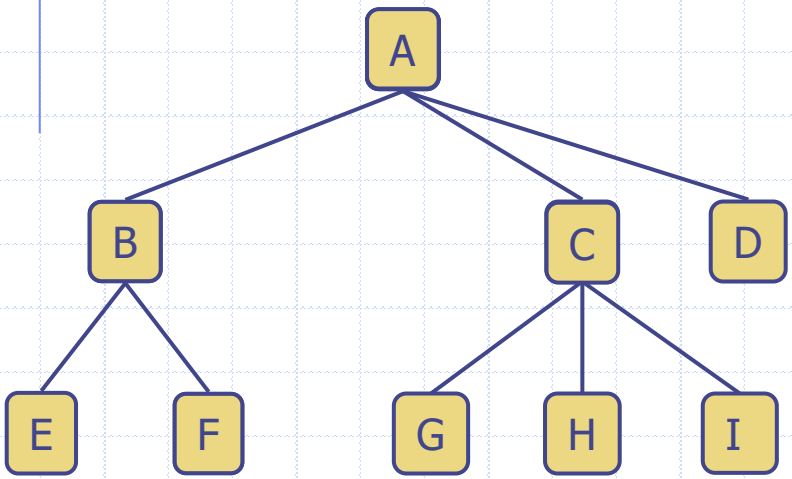
Tree Terminology

- Level
- Depth of a node / tree
- Height of a node / tree
- Subtree



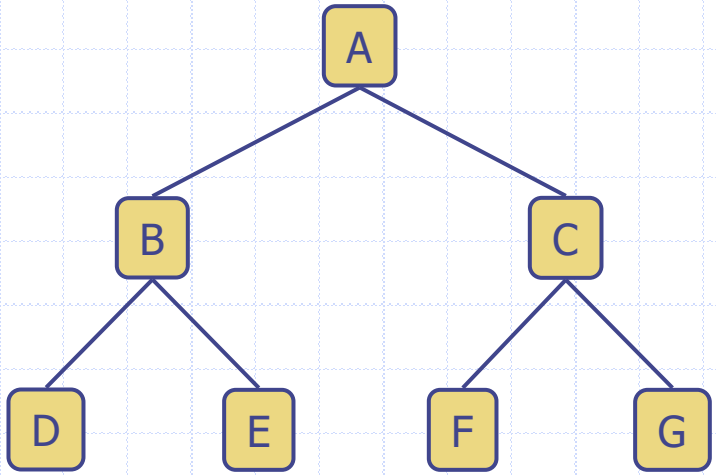
Tree Terminology

- k-ary Tree



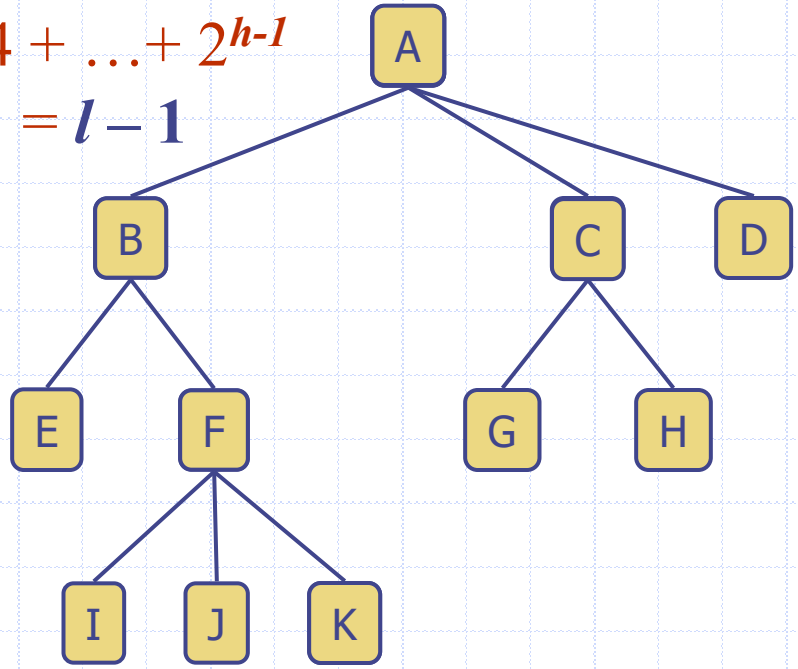
3-ary Tree

- 2-ary Tree
  - binary tree

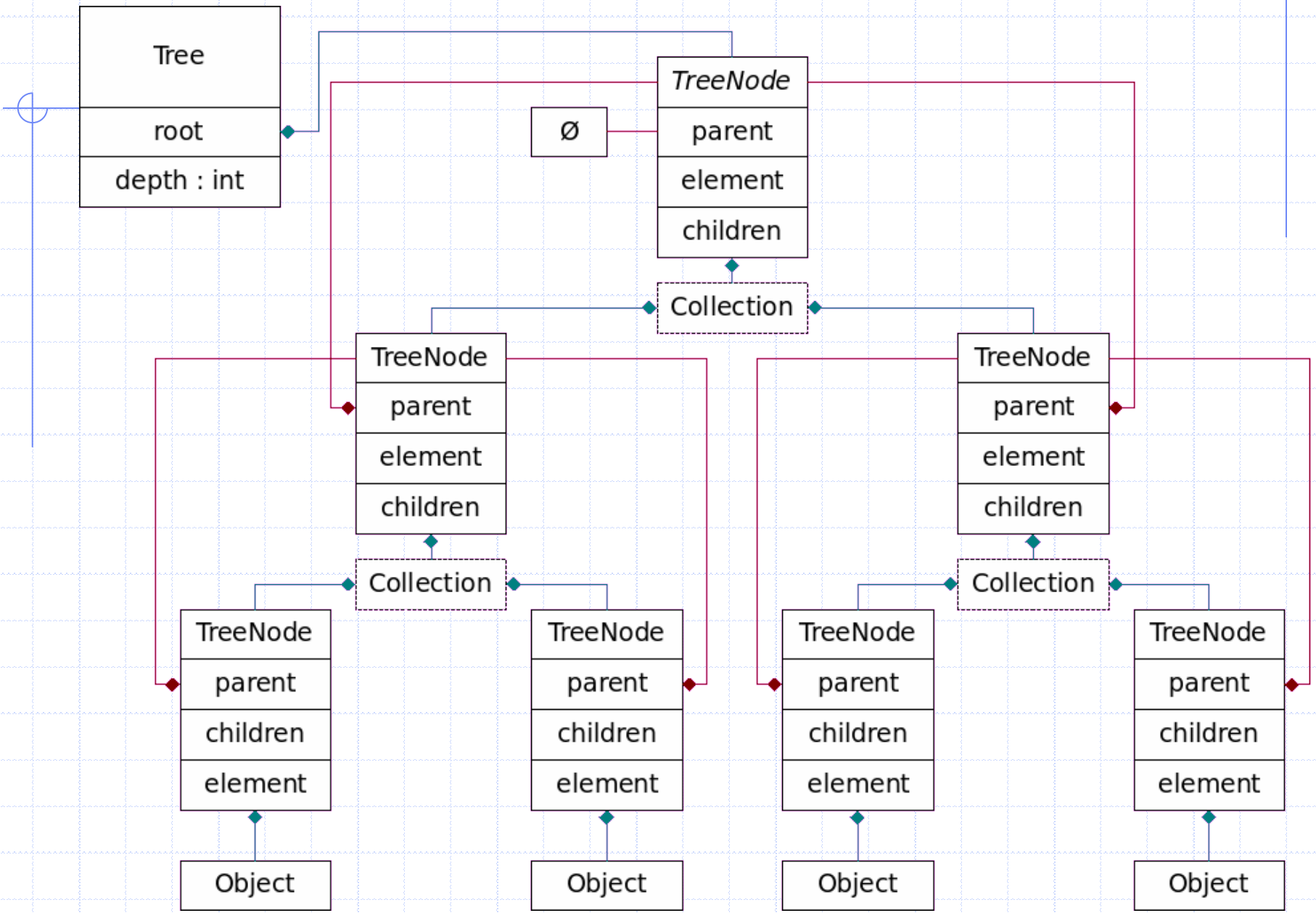


Tree Properties

- If every internal node has at least 2 child nodes
  - if l is number of leaf nodes
  - number of internal nodes is at most $l - 1$
  - $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1 = l - 1$



General Tree Structure



Tree ADT

- General Tree methods
  - size()
  - isEmpty()
  - iterator()
  - positions()

Tree ADT

- Accessor methods
  - root() – returns the root
  - parent(p) – returns the parent of p
  - children(p) – Returns a list of children of p
  - numChildren(p) – Returns number of children of p
- Query methods
  - isInternal(p) – True if p has at least one child
  - isExternal(p) – True if p has no children
  - isRoot(p) – True if p has no parent

Tree CDTs

- Common update methods
  - replace(p, o) – replaces (updates) the (data) element at position p with o
  - addRoot(o) – adds a new root to the tree
  - remove(p) – removes a node from position p, restructures the tree if necessary

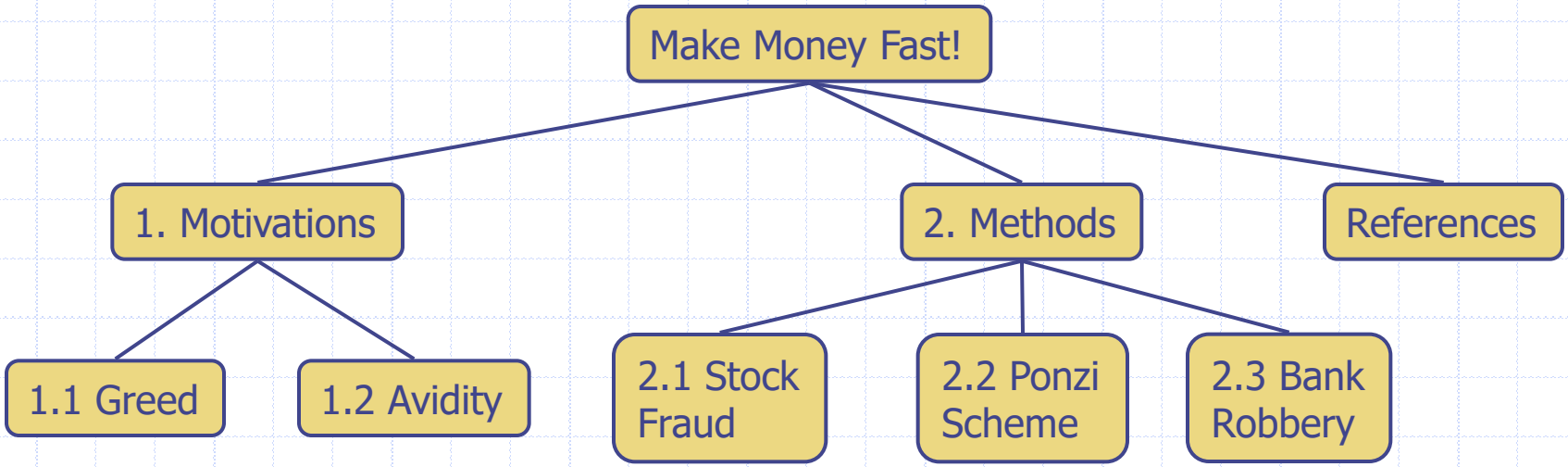
Week 4 – Stacks, Queues, Trees

1. Stacks
2. Queues
3. Tree basics
4. Tree traversals
5. Binary trees

Preorder Traversal

- A node is visited before its descendants
  - visit performs some task at the node
- Application: print a structured document

Algorithm *preOrder*(*p*)
  visit(*p*)
  for each child *c* **of** *p*
    preorder(*c*)

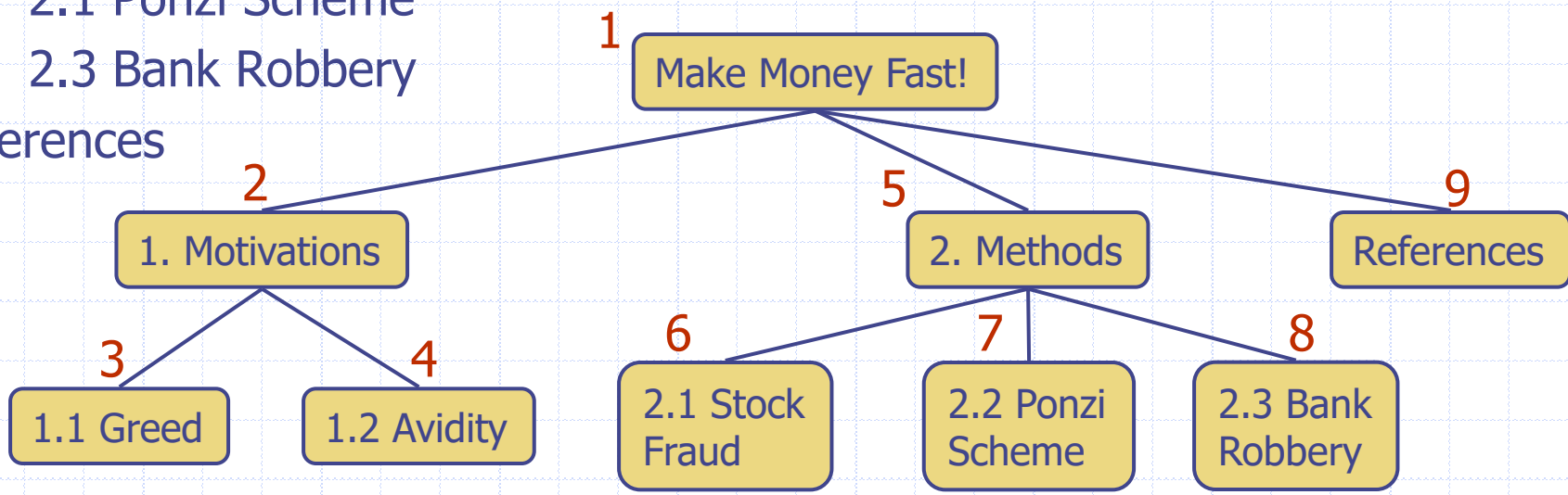


Preorder Traversal

Make Money Fast!

1. Motivations
    - 1.1 Greed
    - 1.2 Avidity
2. Methods
    - 2.1 Stock Fraud
    - 2.1 Ponzi Scheme
    - 2.3 Bank Robbery

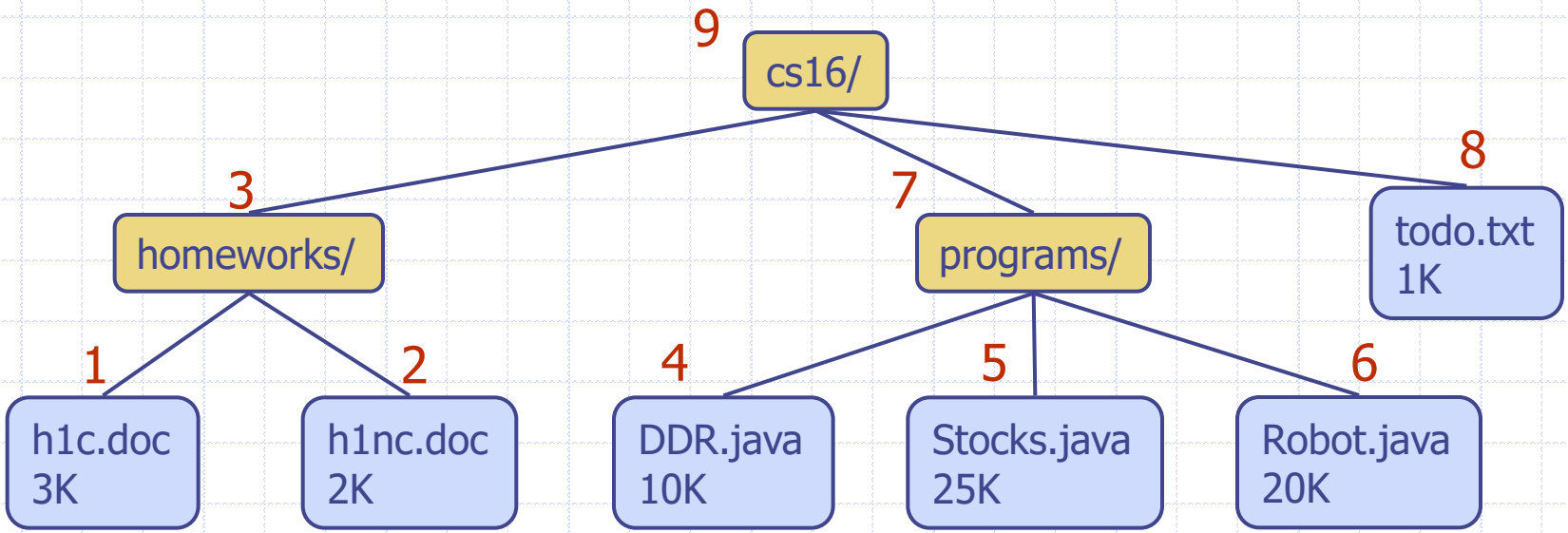
References



Postorder Traversal

- A node is visited after its descendants

Algorithm ***postOrder(p)***
  for each child ***c*** **of** ***p***
    postOrder **(*c*)**
  visit(p)



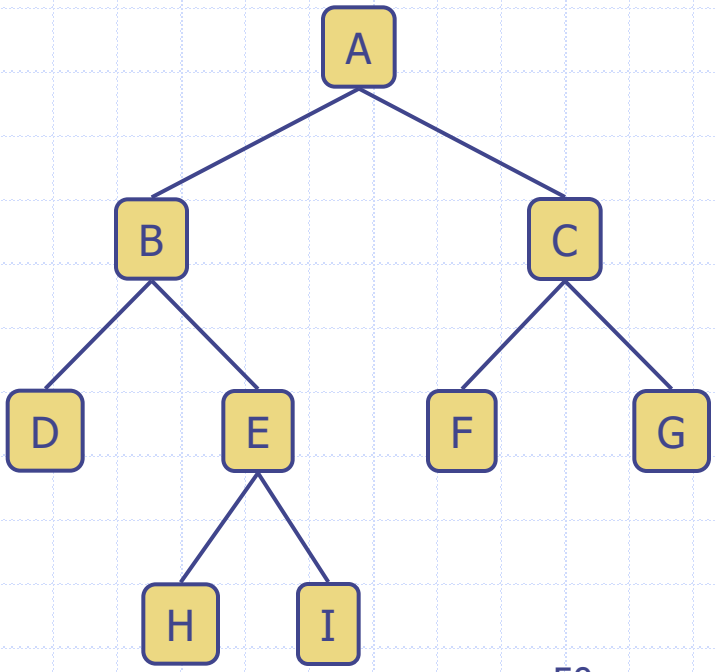
Week 4 – Stacks, Queues, Trees

1. Stacks
2. Queues
3. Tree basics
4. Tree traversals
5. Binary trees

Binary Trees

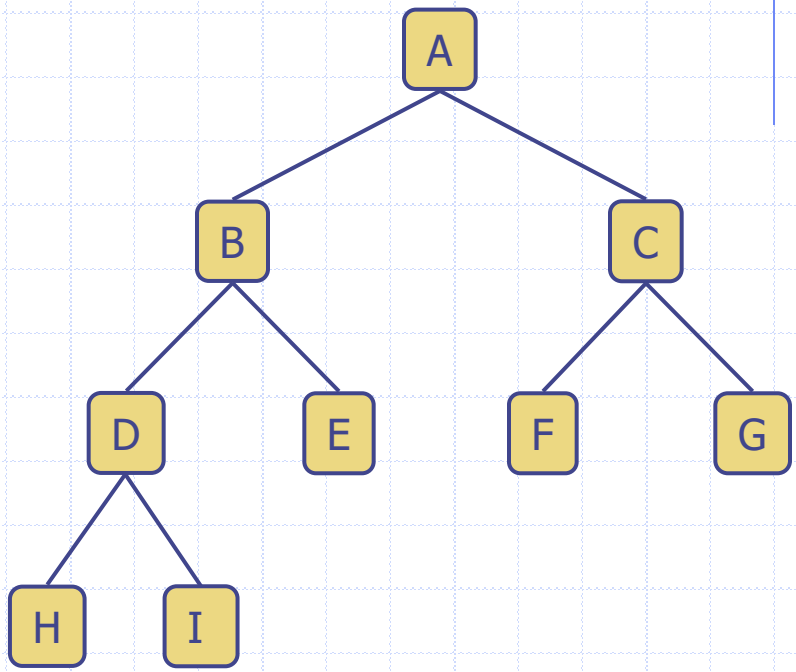
- Each internal node has at most two children
  - exactly two for a **proper** binary trees
- Internal node has a left child and right child
- **Recursive definition:** a binary tree is either
  - an empty tree, or
  - a tree whose root node has an ordered pair of children, each of which is a binary tree

- Applications
  - arithmetic expressions
  - decision processes
  - searching



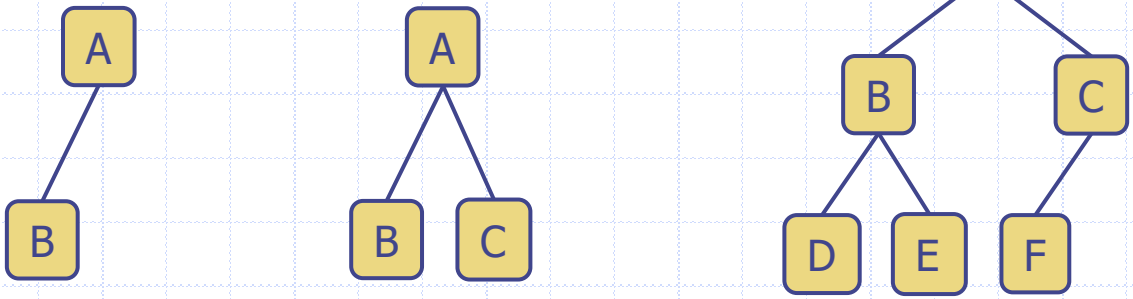
Binary Tree Terminology

- Full level
  - level l is full if it contains 2^l nodes
- Complete binary tree
  - for height h
  - levels 0 … h – 1 are full
  - In level h, all leaf nodes are as far left as possible

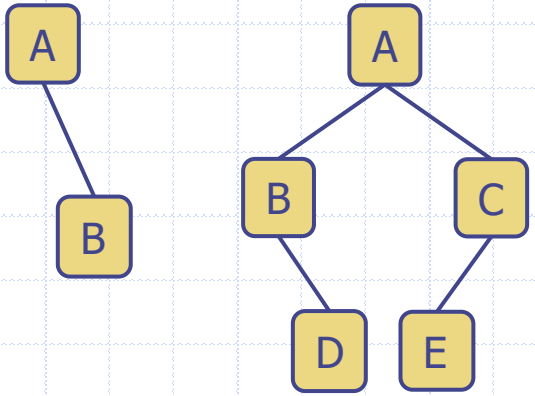


Binary Tree Terminology

- Complete Binary Trees

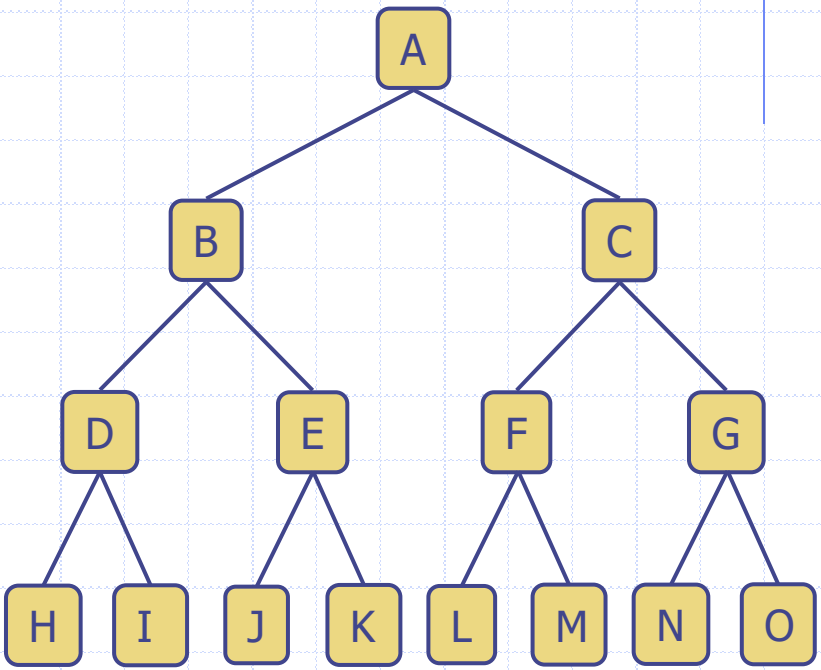


- Not Complete Binary Trees



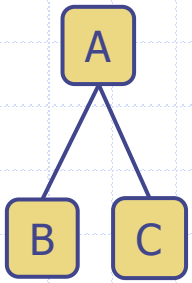
Binary Tree Terminology

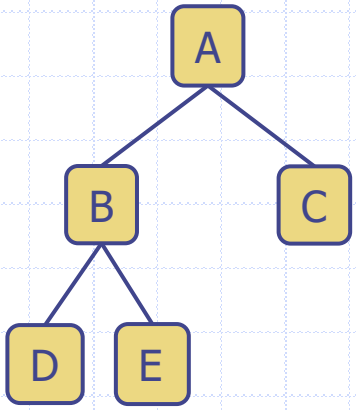
- Proper binary tree
  - every node, *except for leaves*, has two children
  - also called a full binary tree



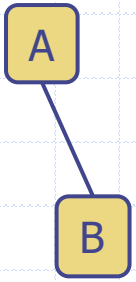
Binary Tree Terminology

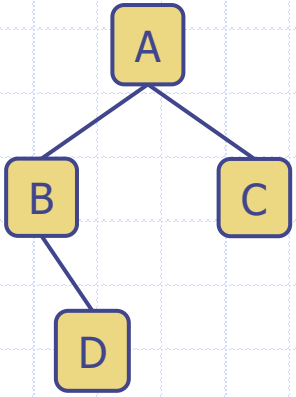
- Proper Binary Trees





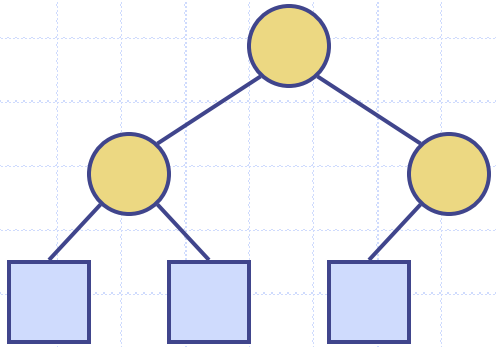
- Not Proper Binary Trees





Properties of Complete Binary Trees

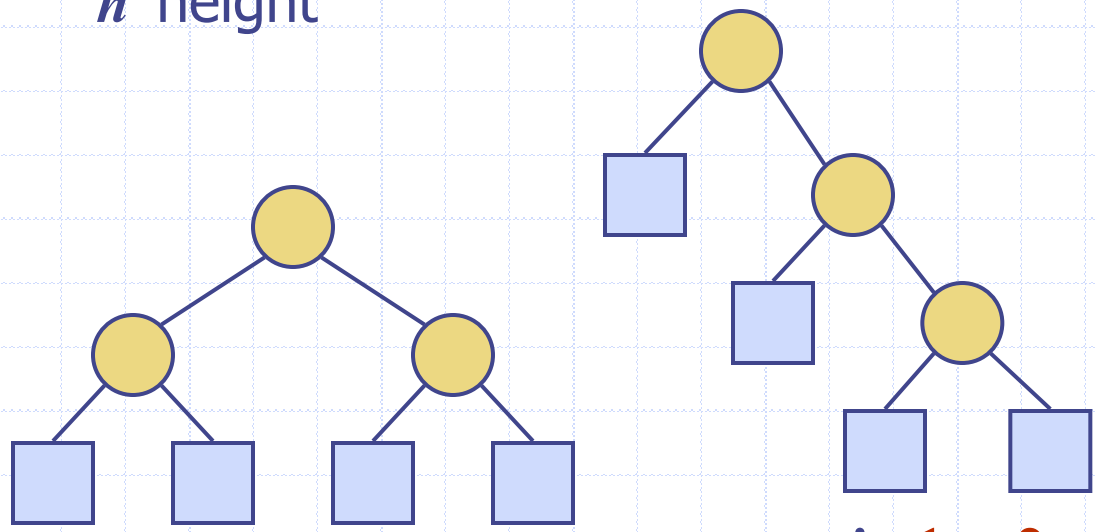
- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height
- Properties
  - $h = O(\log n)$
    - with at least 2 nodes



$$1 + 2 + 4 + \ldots + 2^{h-1}$$
$$= 2^h - 1 <= n$$

Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



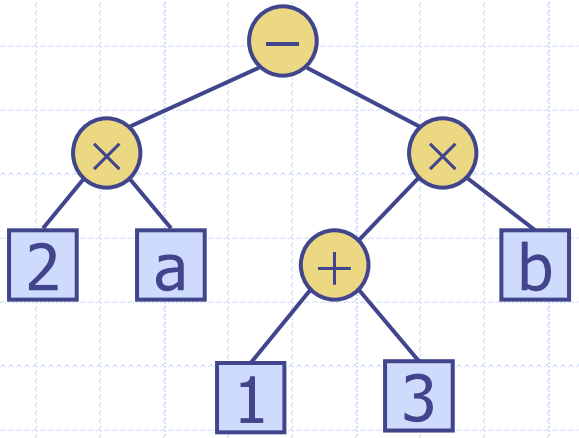
- Properties
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

$i = 1 + 2 + 4 + \ldots + 2^{h-1}$

$= 2^h - 1 = e - 1$

Exercise: Arithmetic Expression Tree

- Binary tree associated with the arithmetic expression $(2 \times a - (1 + 3) \times b)$
  - Count the following:
    - Nodes
    - Roots
    - Leaves
    - External nodes
    - Internal nodes
  - What's the height of the tree?
  - What's the depth of the tree?



Binary Tree ADT

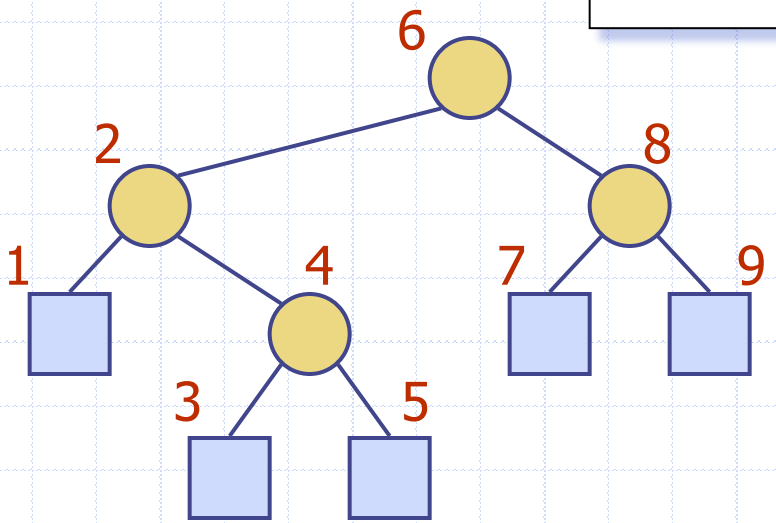
- Extends Tree ADT
  - inherits all methods of the Tree ADT
- Additional methods
  - left(p)
  - right(p)
  - sibling(p)

Inorder Traversal

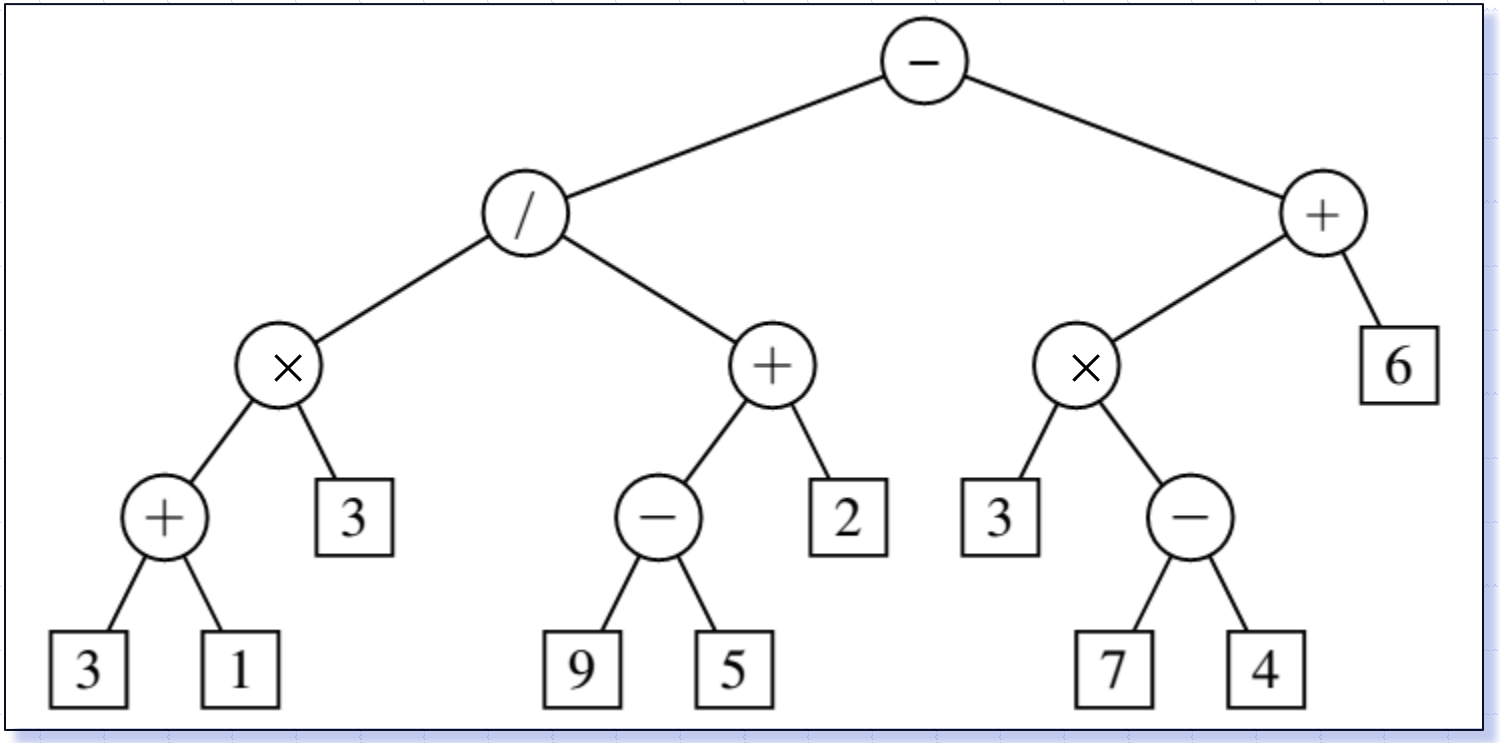
- A node is visited after its left subtree and before its right subtree

Algorithm ***inOrder(p)***

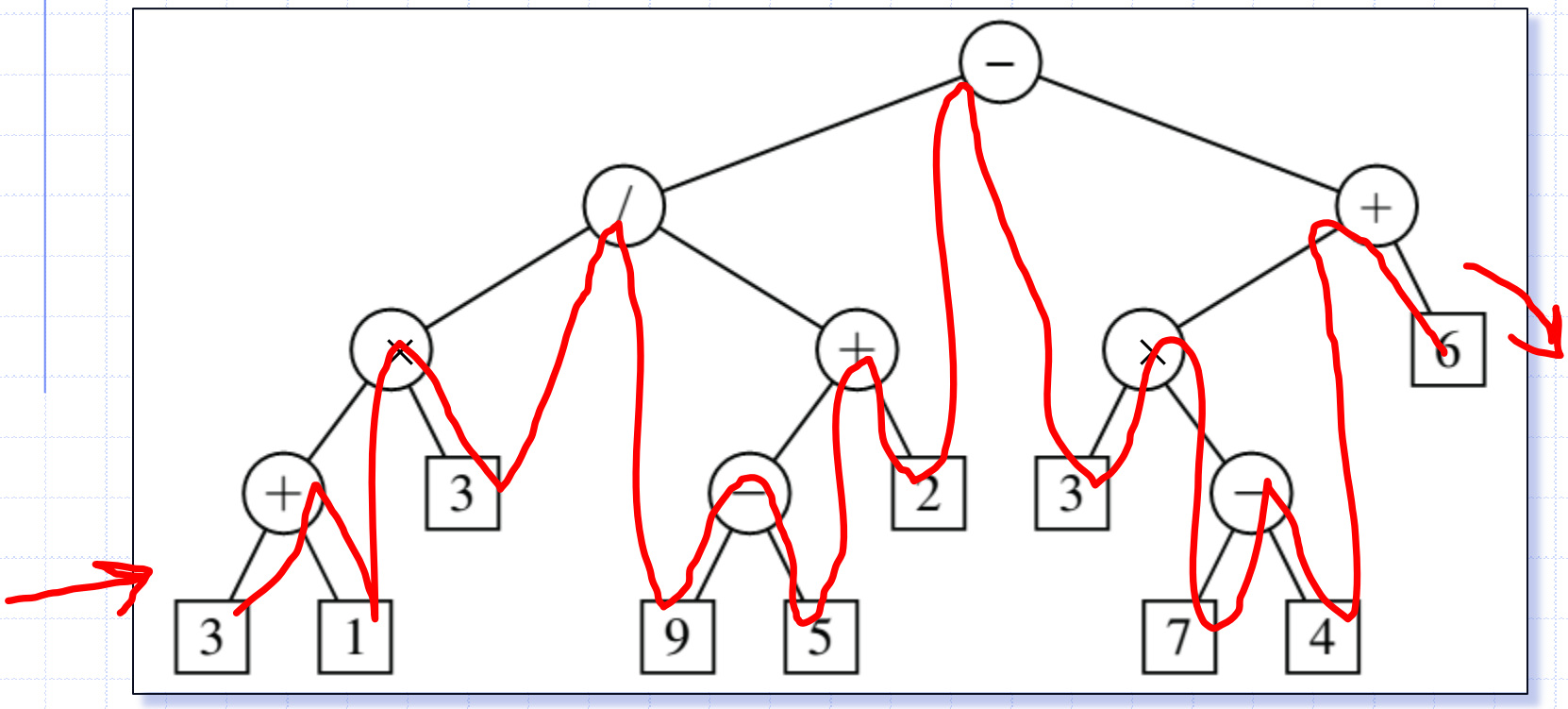
```
if left(p) ≠ null
    inOrder (left(p))
visit(p)
if right(p) ≠ null
    inOrder (right(p))
```



What order would you visit the nodes using an inorder traversal?

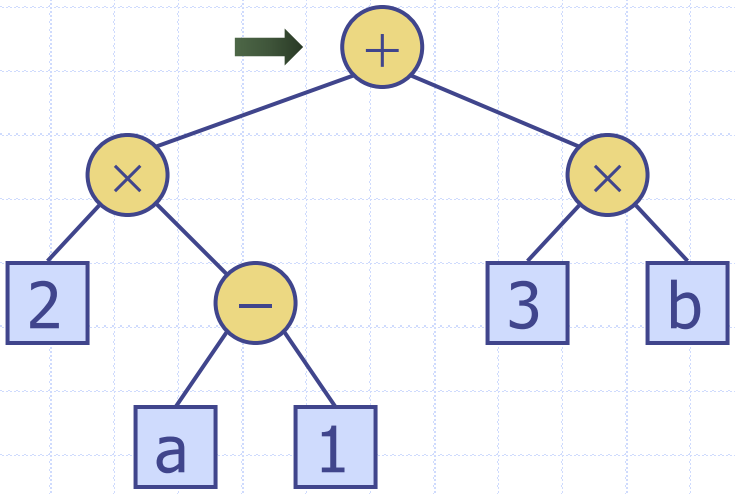


What order would you visit the nodes using an inorder traversal?



Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

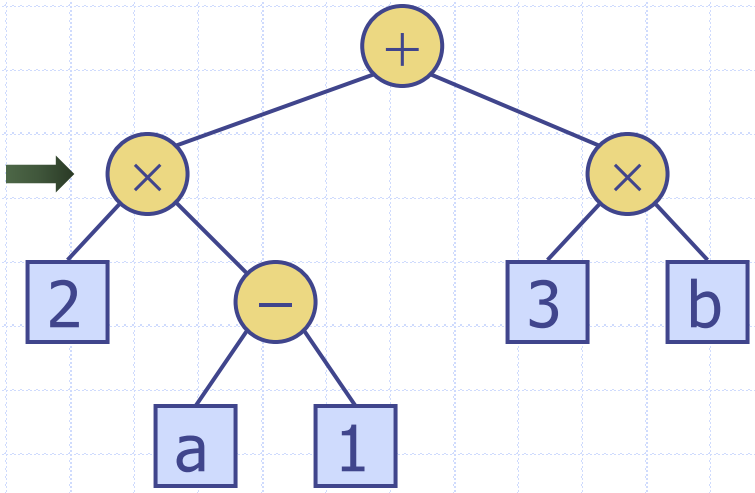


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

(

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

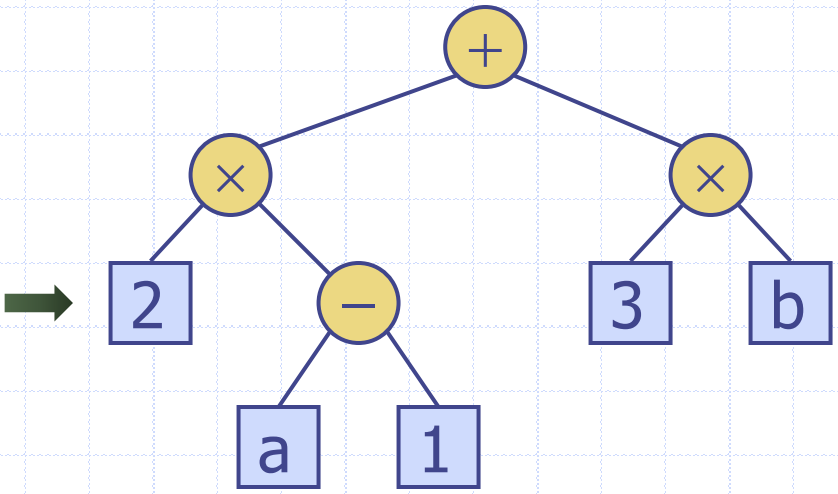


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

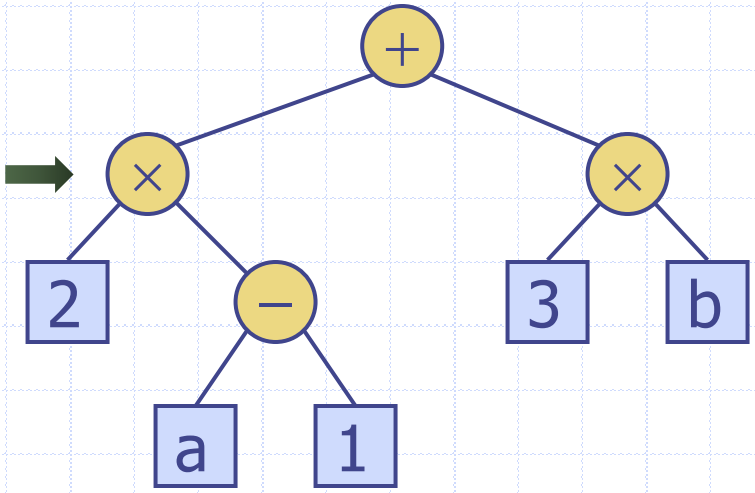


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

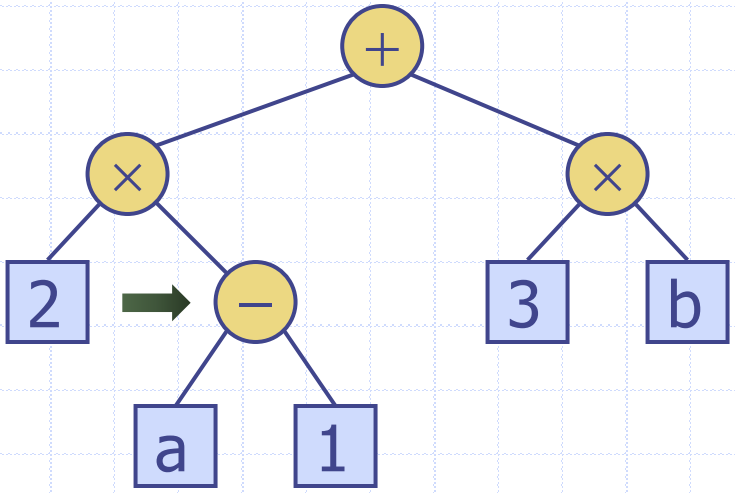


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 ×

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

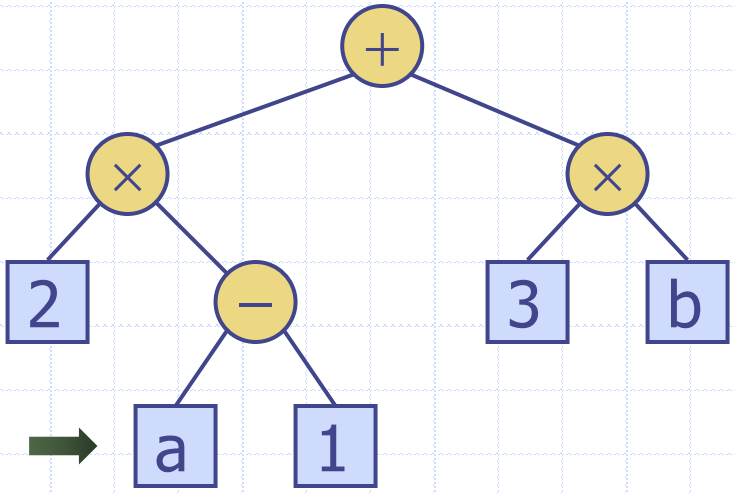


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

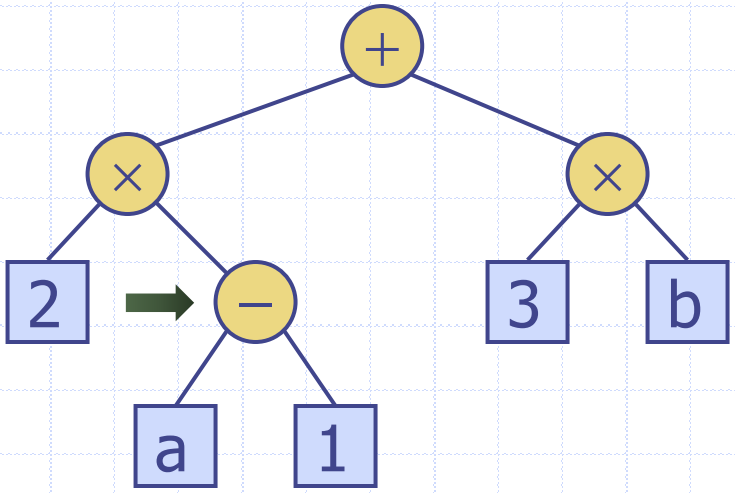


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

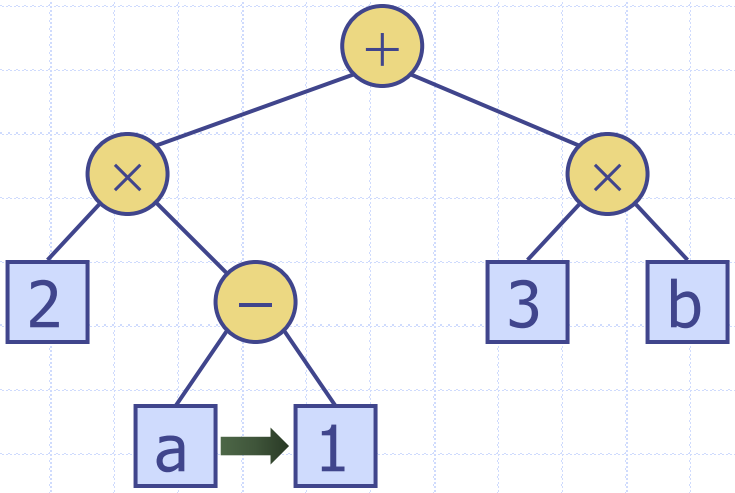


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a −

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

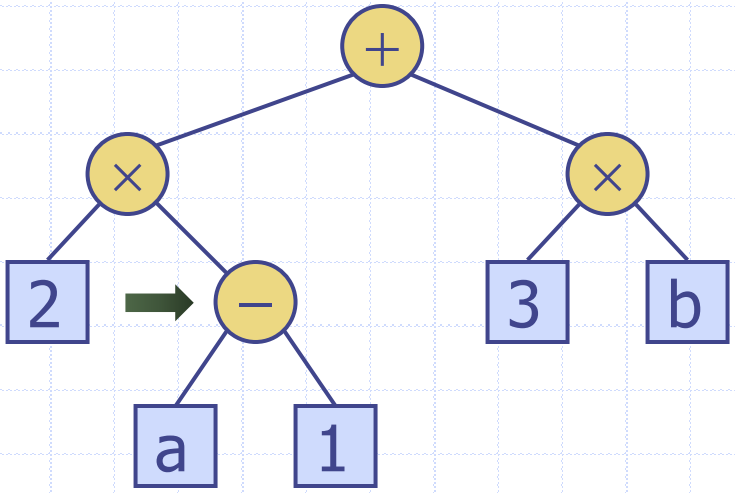


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a – 1

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

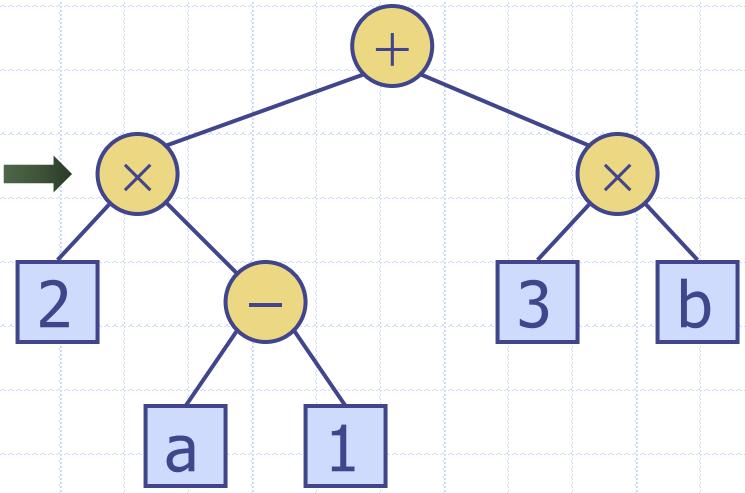


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a – 1)

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

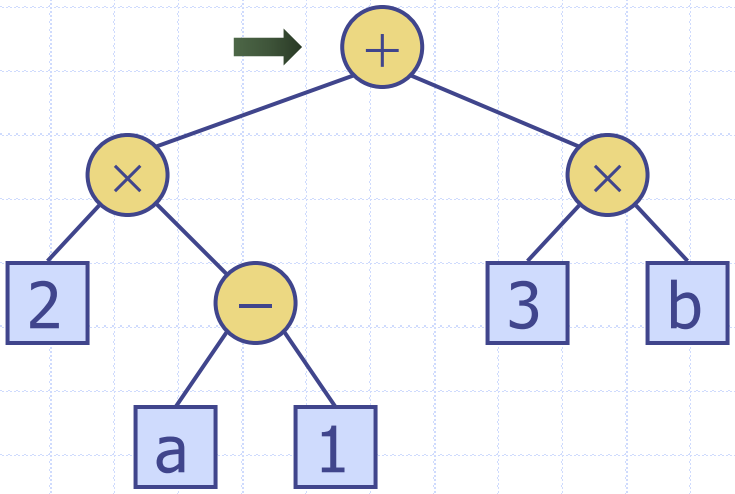


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a − 1))

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

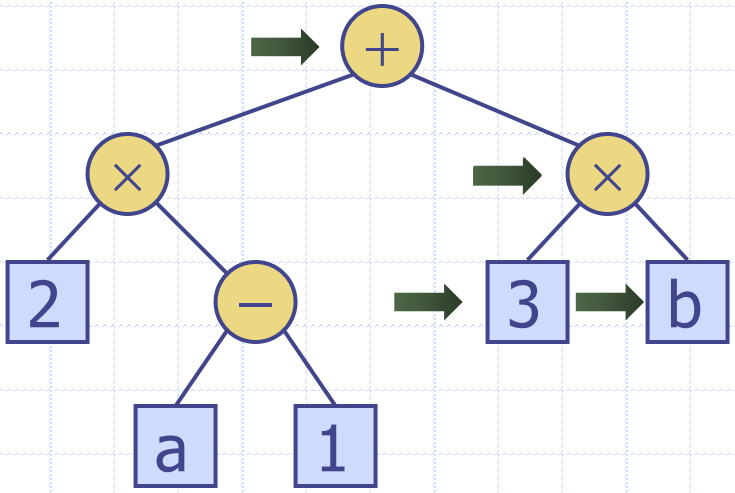


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a − 1)) +

Print Arithmetic Expressions

- Print operand or operator when visiting node
- Print "(" before traversing left subtree
- Print ")" after traversing right subtree

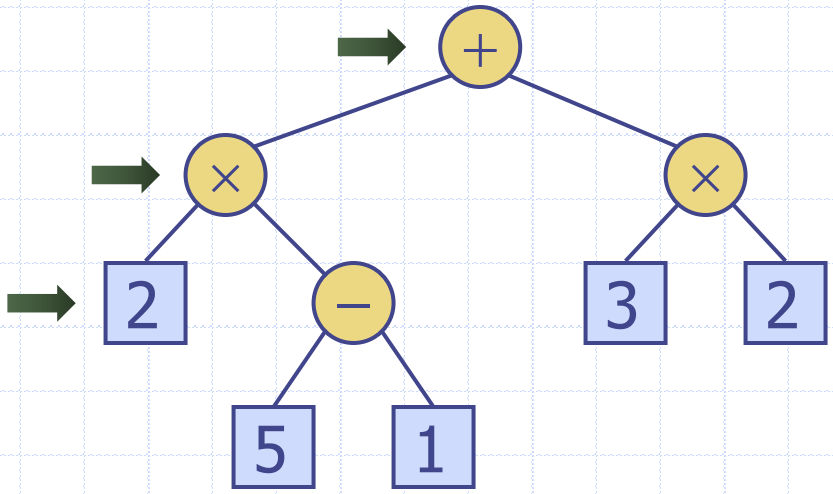


Algorithm *printExpression*(*p*)
  if *left*(*p*) ≠ **null**
    print("(")
    inOrder (*left*(*p*))
  print(*p.getElement*())
  if *right*(*p*) ≠ **null**
    inOrder (*right*(*p*))
    print (")")

((2 × (a − 1)) + (3 × b))

Evaluate Arithmetic Expressions

- Recursive method returning the value of a subtree
- When visiting an internal node, combine the values of the subtrees

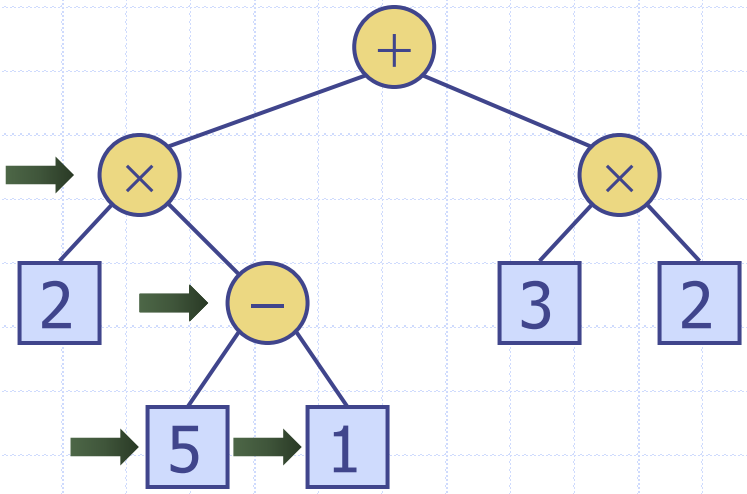


Algorithm *evalExpr*(*p*)
  if *isExternal*(*p*)
    return *p.getElement*()
  else
    $x \leftarrow$ *evalExpr*(*left*(*p*))
    $y \leftarrow$ *evalExpr*(*right*(*p*))
    $\lozenge \leftarrow$ operator stored at *p*
    return $x \lozenge y$

$$x = \begin{cases} x = 2 \end{cases}$$

Evaluate Arithmetic Expressions

- Recursive method returning the value of a subtree
- When visiting an internal node, combine the values of the subtrees

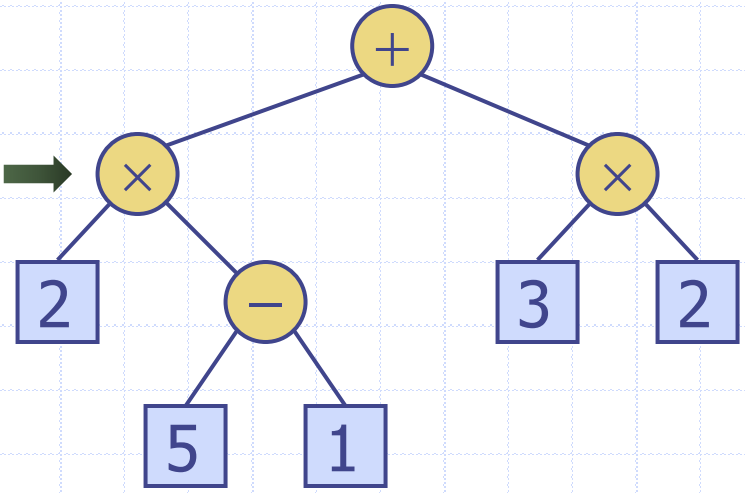


Algorithm *evalExpr*(*p*)
  if *isExternal*(*p*)
    return *p.getElement*()
  else
    $x \leftarrow$ *evalExpr*(*left*(*p*))
    $y \leftarrow$ *evalExpr*(*right*(*p*))
    $\lozenge \leftarrow$ operator stored at *p*
    return $x \lozenge y$

$$x = \begin{cases} x = 2 & \\ x = 5 & \\ y = 1 & \\ \lozenge = - & \end{cases}$$

Evaluate Arithmetic Expressions

- Recursive method returning the value of a subtree
- When visiting an internal node, combine the values of the subtrees

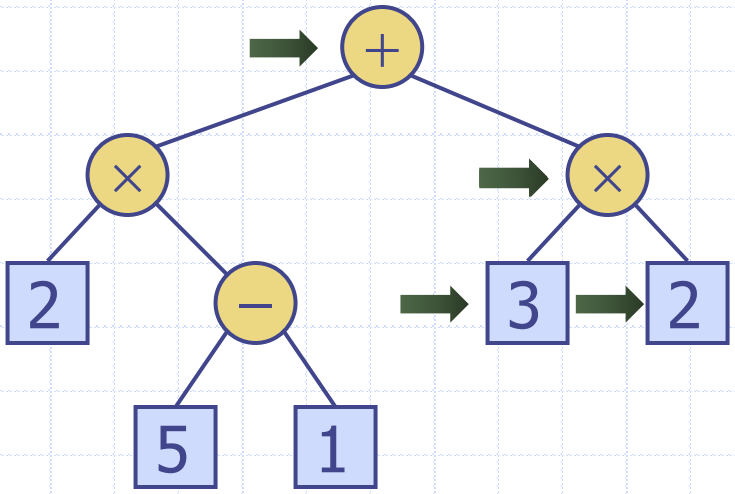


Algorithm *evalExpr*(*p*)
  if *isExternal*(*p*)
    return *p.getElement*()
  else
    $x \leftarrow$ *evalExpr*(*left*(*p*))
    $y \leftarrow$ *evalExpr*(*right*(*p*))
    $\lozenge \leftarrow$ operator stored at *p*
    return $x \lozenge y$

$$x = \begin{cases} x = 2 \\ y = 4 \\ \lozenge = \times \end{cases}$$

Evaluate Arithmetic Expressions

- Recursive method returning the value of a subtree
- When visiting an internal node, combine the values of the subtrees

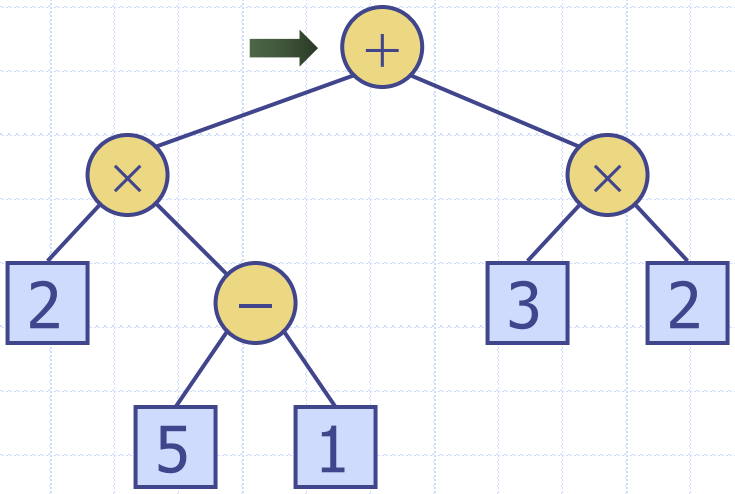


Algorithm *evalExpr*(*p*)
  if *isExternal*(*p*)
    return *p.getElement*()
  else
    $x \leftarrow$ *evalExpr*(*left*(*p*))
    $y \leftarrow$ *evalExpr*(*right*(*p*))
    $\lozenge \leftarrow$ operator stored at *p*
    return $x \lozenge y$

$x = 8$
$y = 3 \times 2$

Evaluate Arithmetic Expressions

- Recursive method returning the value of a subtree
- When visiting an internal node, combine the values of the subtrees

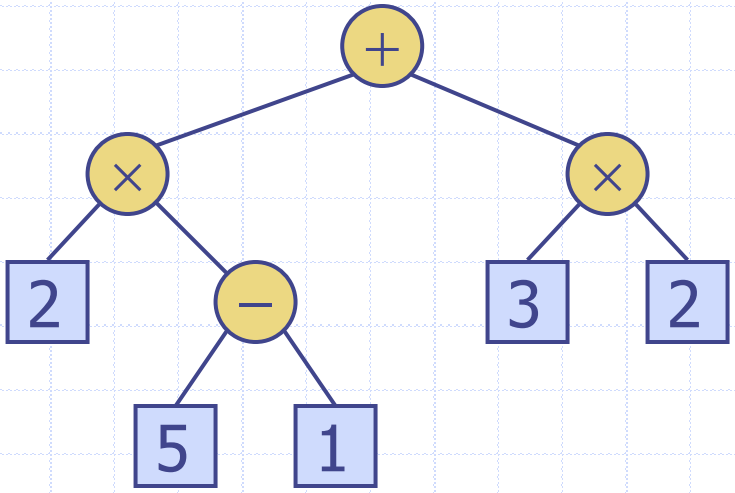


Algorithm *evalExpr*(*p*)
  if *isExternal*(*p*)
    return *p.getElement*()
  else
    $x \leftarrow$ *evalExpr*(*left*(*p*))
    $y \leftarrow$ *evalExpr*(*right*(*p*))
    $\lozenge \leftarrow$ operator stored at *p*
    return $x \lozenge y$

$x = 8$
$y = 3 \times 2$
$\lozenge = +$

Evaluate Arithmetic Expressions

- Recursive method returning the value of a subtree
- When visiting an internal node, combine the values of the subtrees

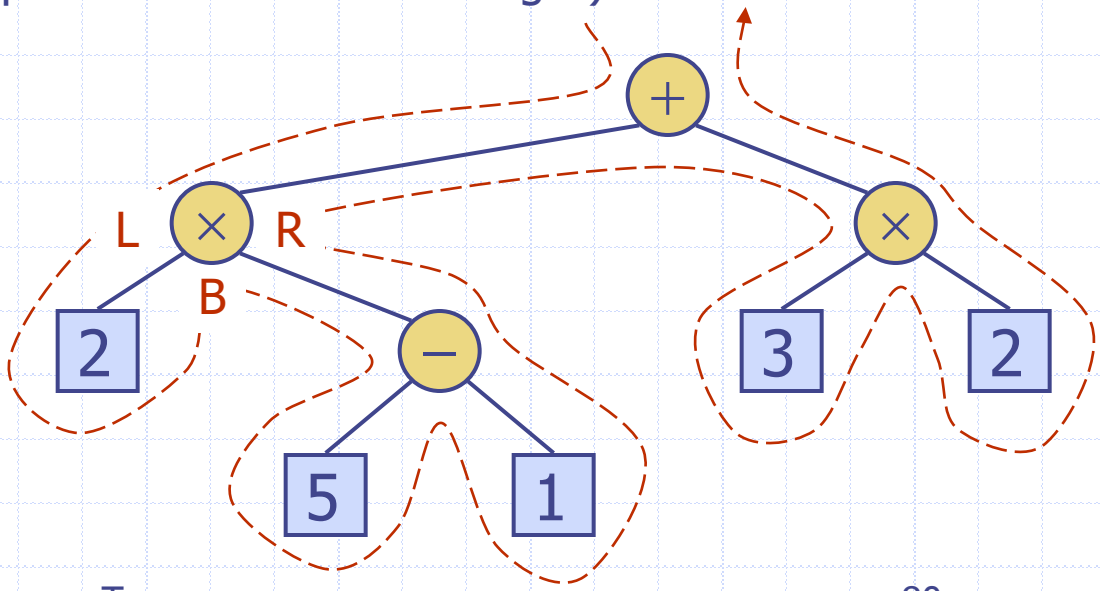


Algorithm *evalExpr*(*p*)
  if *isExternal*(*p*)
    return *p.getElement*()
  else
    $x \leftarrow$ *evalExpr*(*left*(*p*))
    $y \leftarrow$ *evalExpr*(*right*(*p*))
    $\lozenge \leftarrow$ operator stored at *p*
    return $x \lozenge y$

14

Euler Tour Traversal

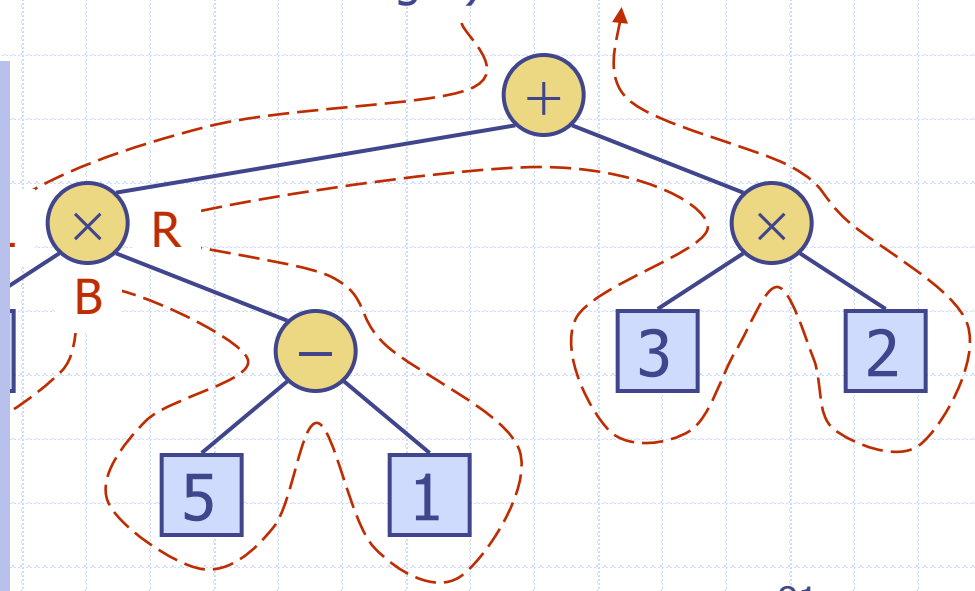
- **Generic** traversal of a binary tree
  - Handles preorder, postorder and inorder traversals
  - Each node (in a **binary** tree) is visited three times
    - Previsit (when we pass the node on its left)
    - Invisit (when we pass the node from below)
    - Postvisit (when we pass the node from its right)



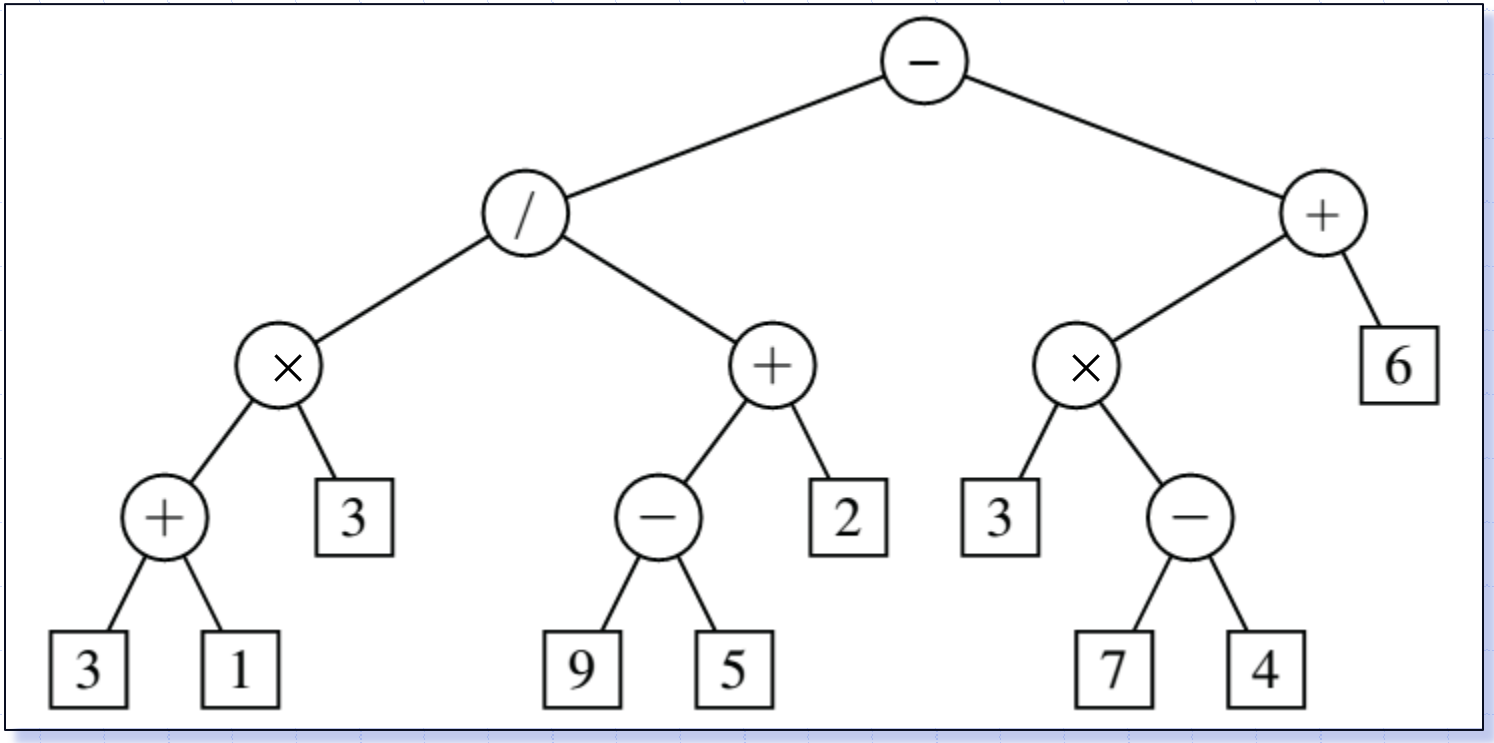
Euler Tour Traversal

- **Generic** traversal of a binary tree
  - Handles preorder, postorder and inorder traversals
  - Each node (in a **binary** tree) is visited three times
    - Previsit (when we pass the node on its left)
    - Invisit (when we pass the node from below)
    - Postvisit (when we pass the node from its right)

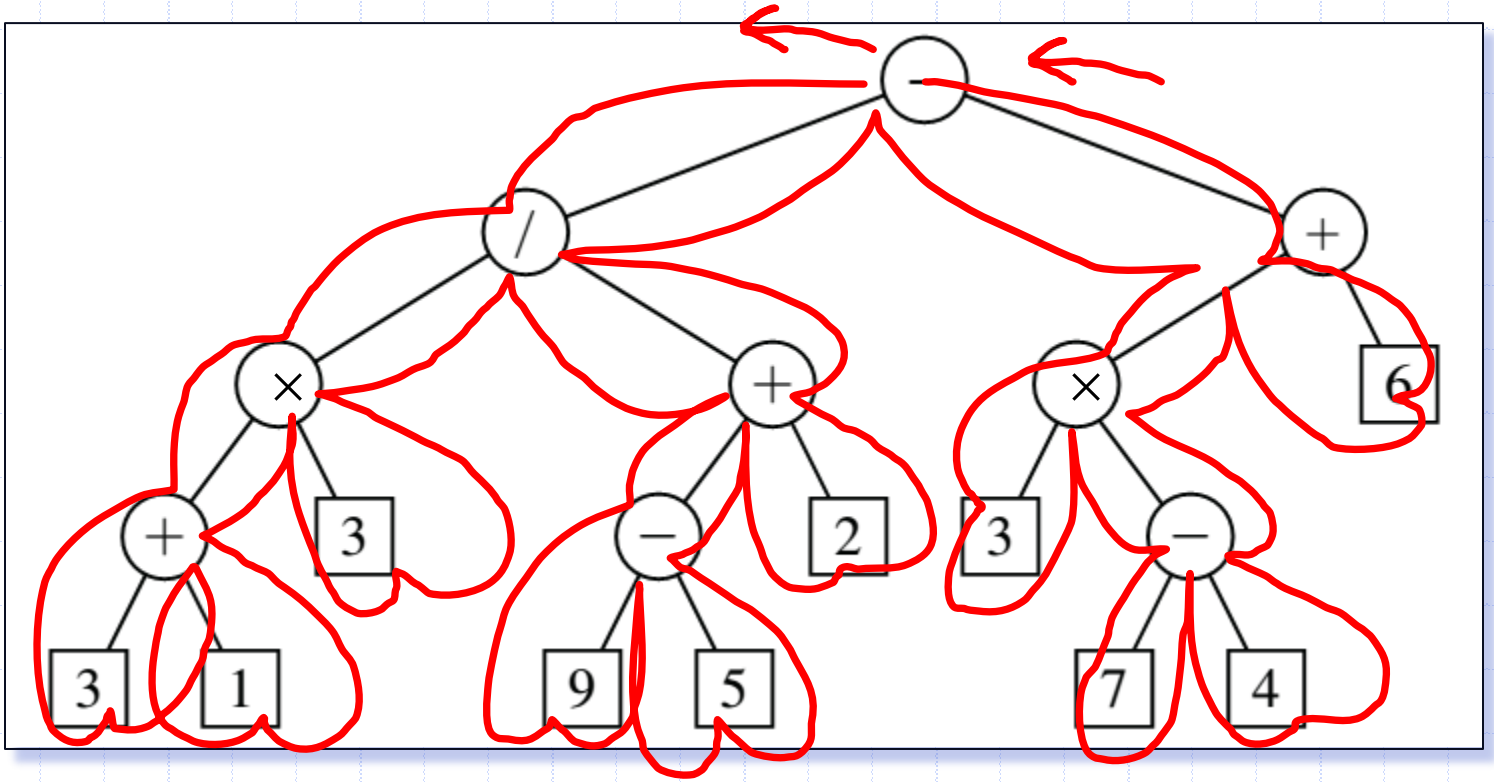
Idea is that we can "blend" different traversals to achieve more complex functionality by "hooking" specific functionality into different visits to each node



What order would you visit the nodes using an Euler tour traversal?



What order would you visit the nodes using an Euler tour traversal?



Evaluating Arithmetic Expression

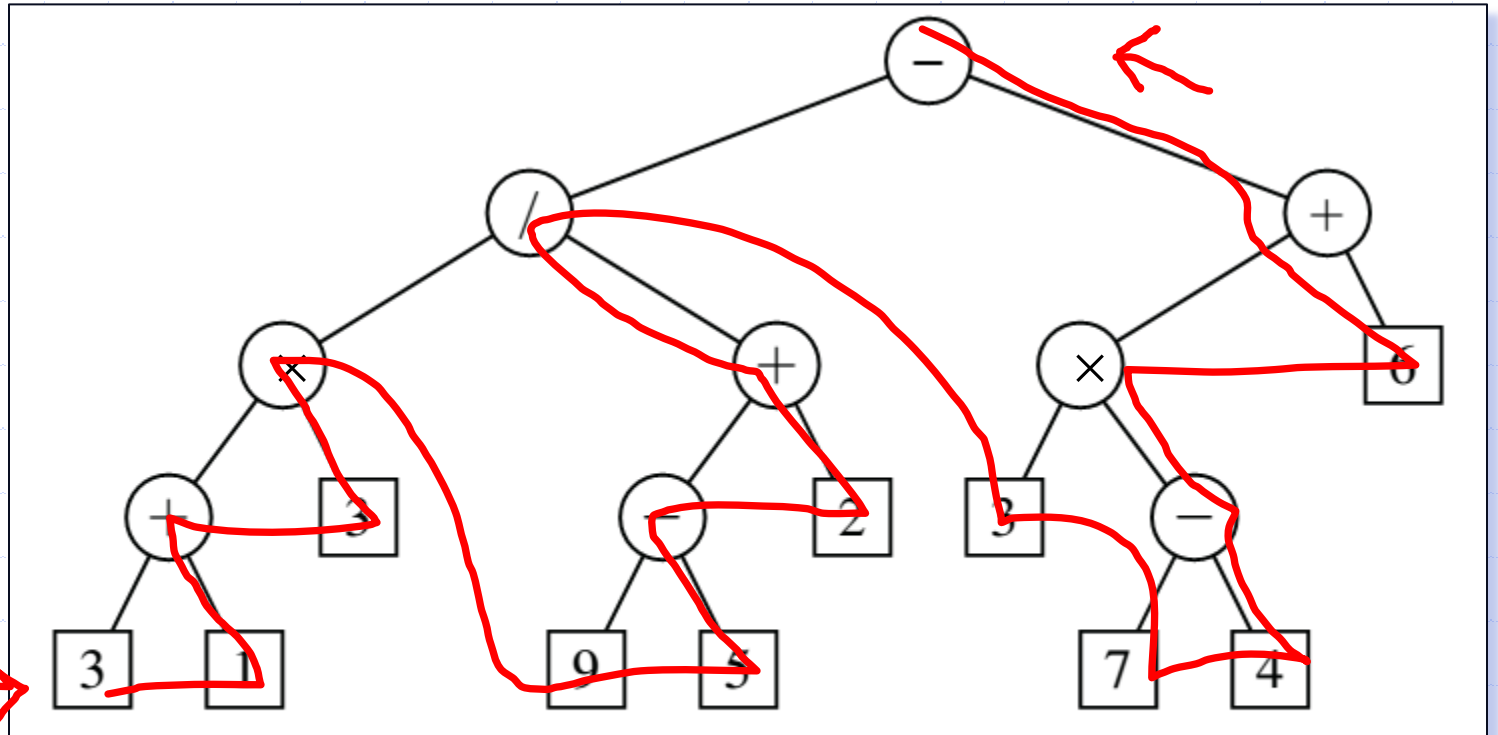
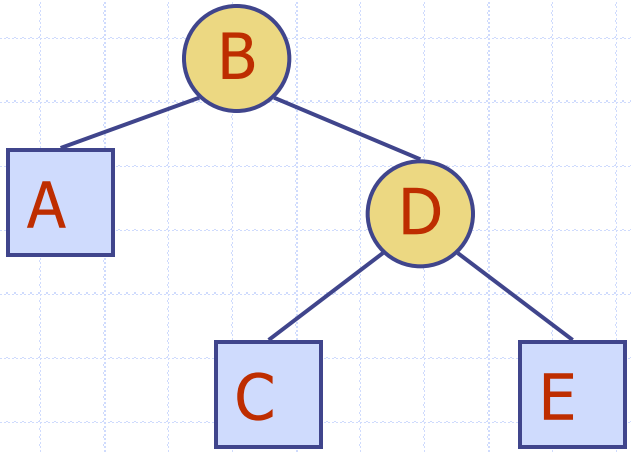
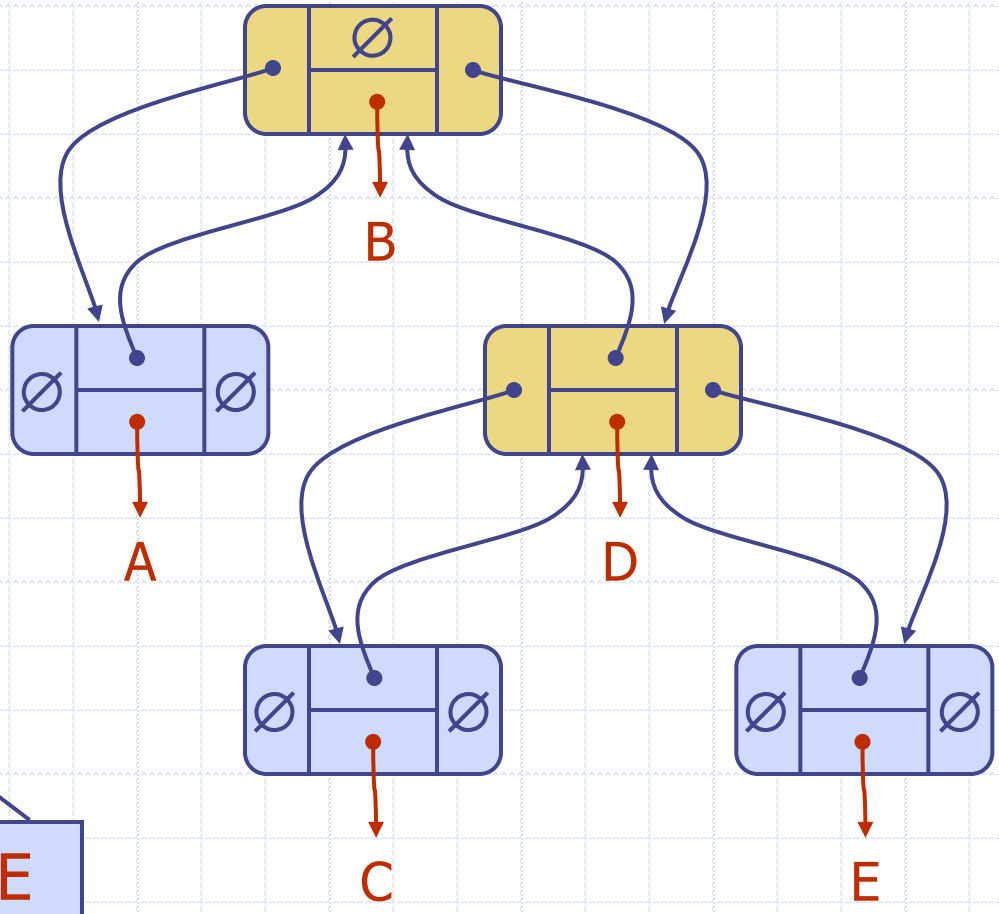


Figure 8.6: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

Linked Structure for Binary Trees

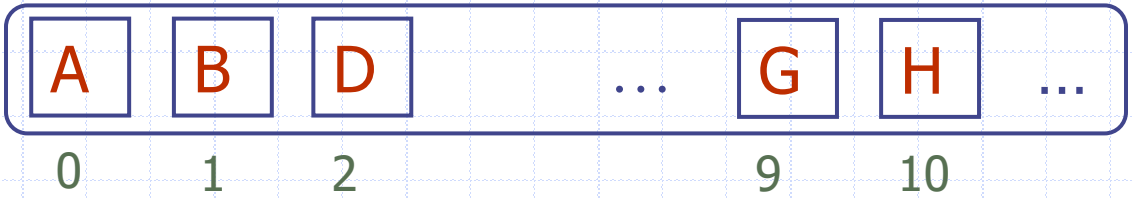
- Node stores
  - element
  - parent node
  - left child node
  - right child node

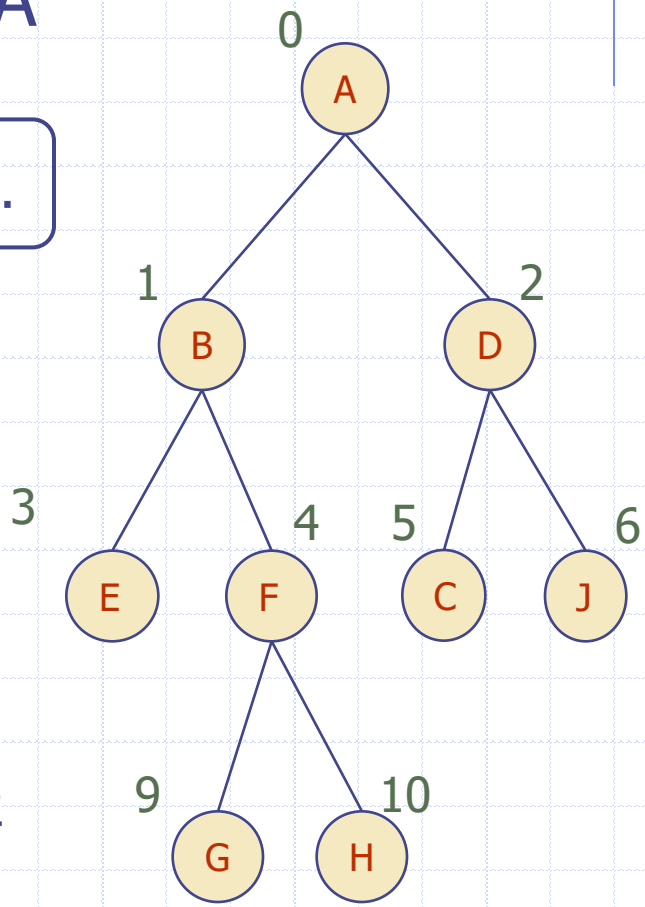




Array-Based Representation of Binary Trees

- Nodes are stored in an array A





Node v is stored at A[rank(v)]

- rank(root) = 0
- if node is the left child of parent(node),
  rank(node) = 2 × rank(parent(node)) + 1
- if node is the right child of parent(node),
  rank(node) = 2 ×rank(parent(node)) + 2

Breadth-First Traversal

- Visit all nodes at depth d before visiting nodes at depth $d + 1$

```
Algorithm breadthFirst(p)
    Q ← new empty queue
    Q.enqueue(p)
    while ¬Q.isEmpty() do
        p = Q.dequeue()
        visit(p)
        for each child c in children(p) do
            Q.enqueue(c)
```

Breadth-First Traversal Example

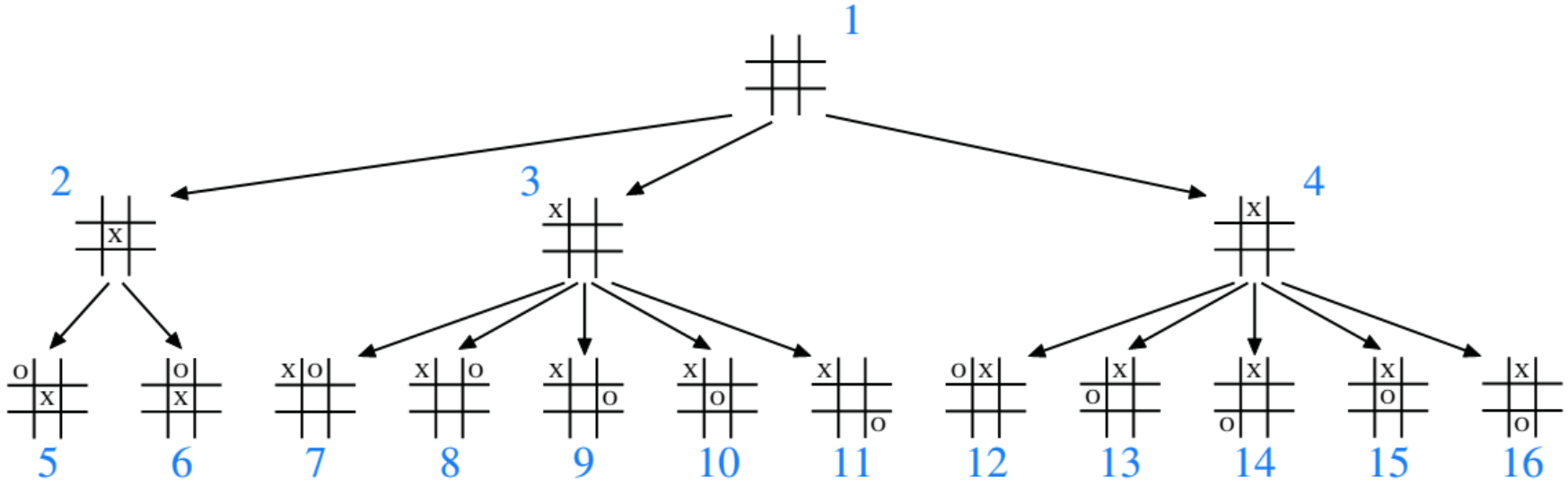
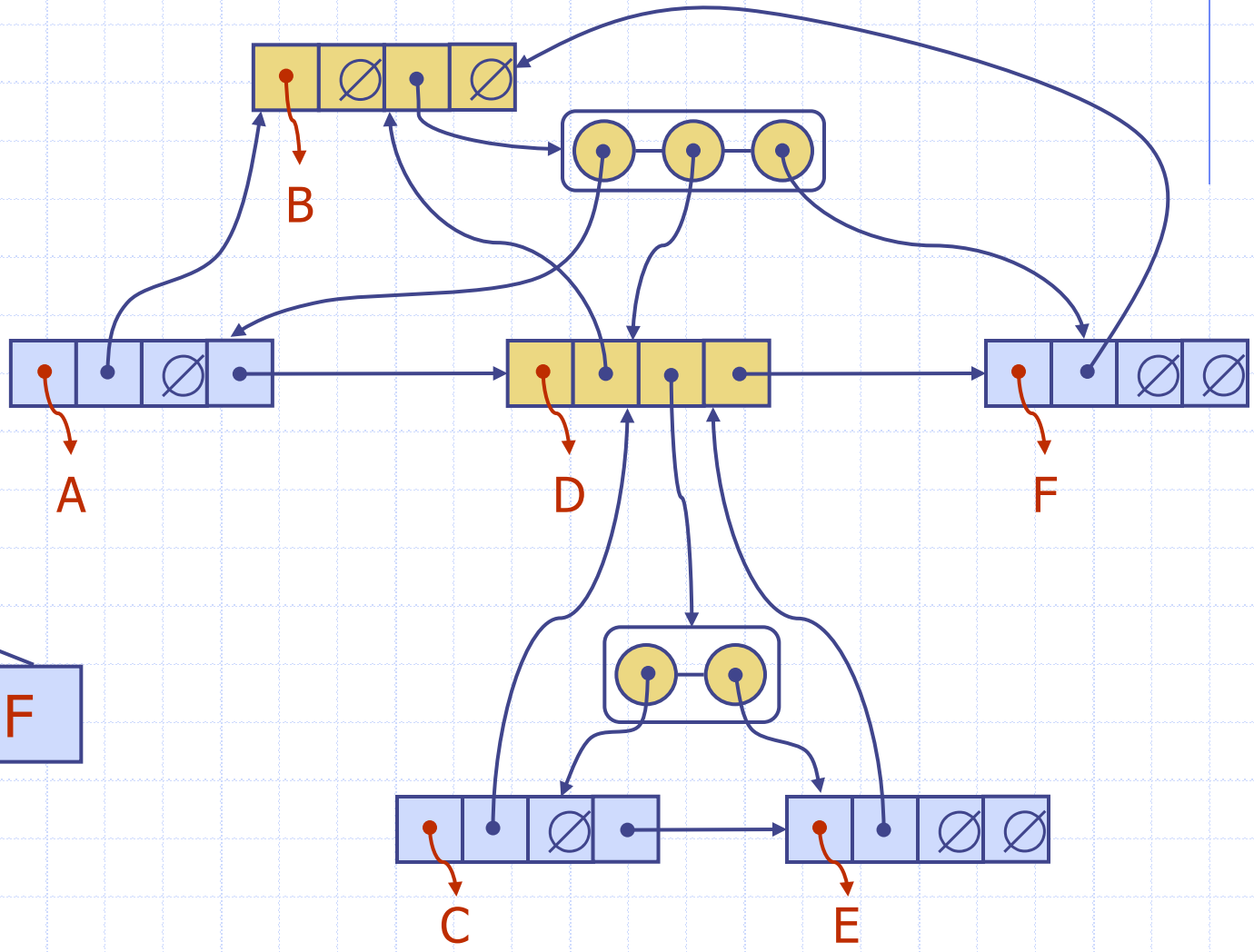


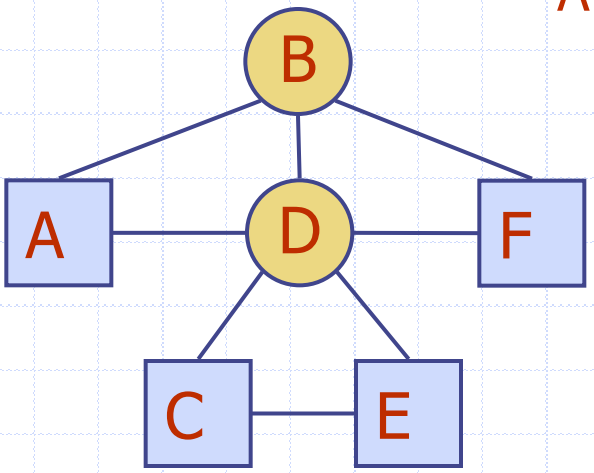
Figure 8.15: Partial game tree for Tic-Tac-Toe when ignoring symmetries; annotations denote the order in which positions are visited in a breadth-first tree traversal.

Breadth-First Traversal & Tree Implementation

- Array
  - Simple linear iteration through array starting at index of the root of the sub-tree
  - Requires each node to have a fixed, small number of children
- Linked Tree Structure

Linked Structure for Breadth-First Traversal





Further Reading

- Data Structures and Algorithms in Python
  - Chapter 1.8
  - Chapter 6
  - Chapter 8
- Introduction to Algorithms
  - Chapter 10.4