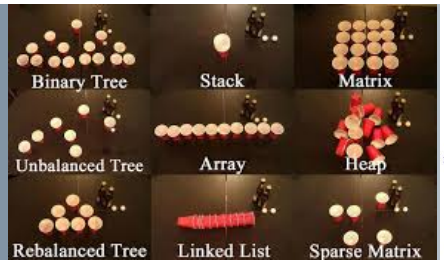
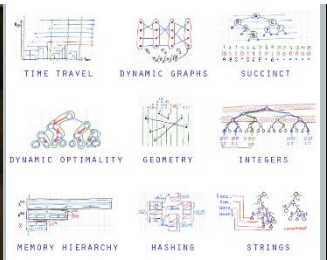
Introduction to Computer Science

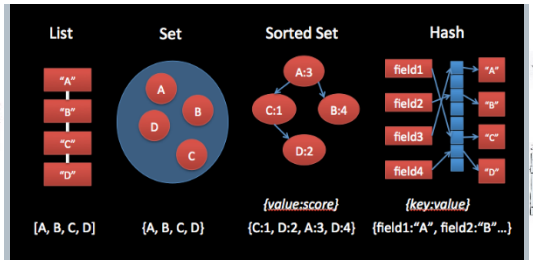
William Hsu
Advanced Computation Laboratory
Department of Computer Science and Engineering
Department of Environmental Biology and Fisheries Science
National Taiwan Ocean University

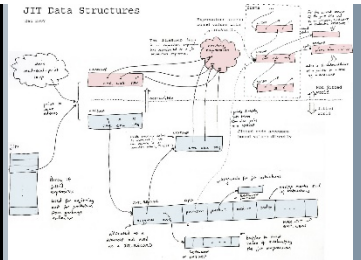
Chapter 8: Data Abstractions

LIFO (Last In, First Out) Also called a "stack".
FIFO (First In, First Out) Also called a "queue".
FISH (First In, Still Here) Also called a "hung printer".
FIGL (First In, Got Lost) Also called a bureaucracy.
LIGL (Last In, Got Lost) Also called a "streams module".
FIGO (First In, Garbage Out) Also called a "random number generator".
GHNW (Got Here, Now What?) Also called a "longjmp botch".







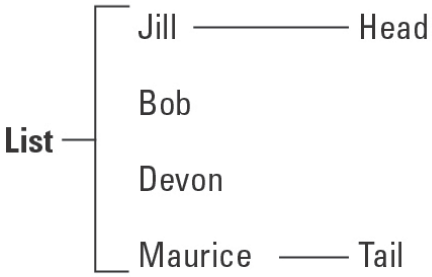


- 8.1 Basic Data Structures
- 8.2 Related Concepts
- 8.3 Implementing Data Structures
- 8.4 A Short Case Study
- 8.5 Customized Data Types
- 8.6 Classes and Objects
- 8.7 Pointers in Machine Language

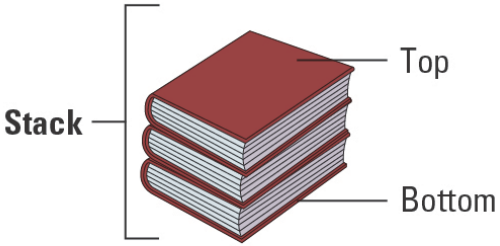
Basic Data Structures

- Arrays
- Aggregates
- List
  - Stack
  - Queue
- Tree

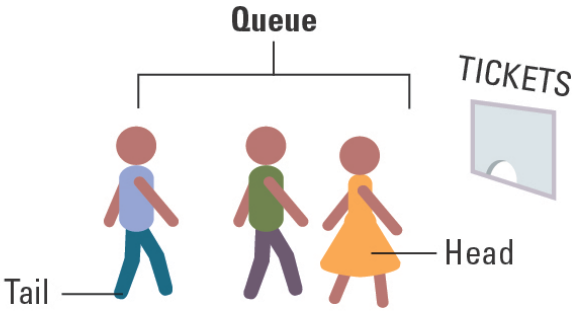
Lists, Stacks, and Queues



a. A list of names



b. A stack of books



c. A queue of people

Terminology for Arrays

- **Array**: A block of data whose entries are of same type.
- A two **dimensional** array consists for rows and columns.
- **Indices** are used to identify positions.

Terminology for Aggregates

- **Aggregate**: A block of data items that might be of different type or sizes.
- Each data item is called a **field**.
- Fields are usually accessed by name.

Terminology for Lists

- **List:** A collection of data whose entries are arranged sequentially.
- **Head:** The beginning of the list.
- **Tail:** The end of the list.

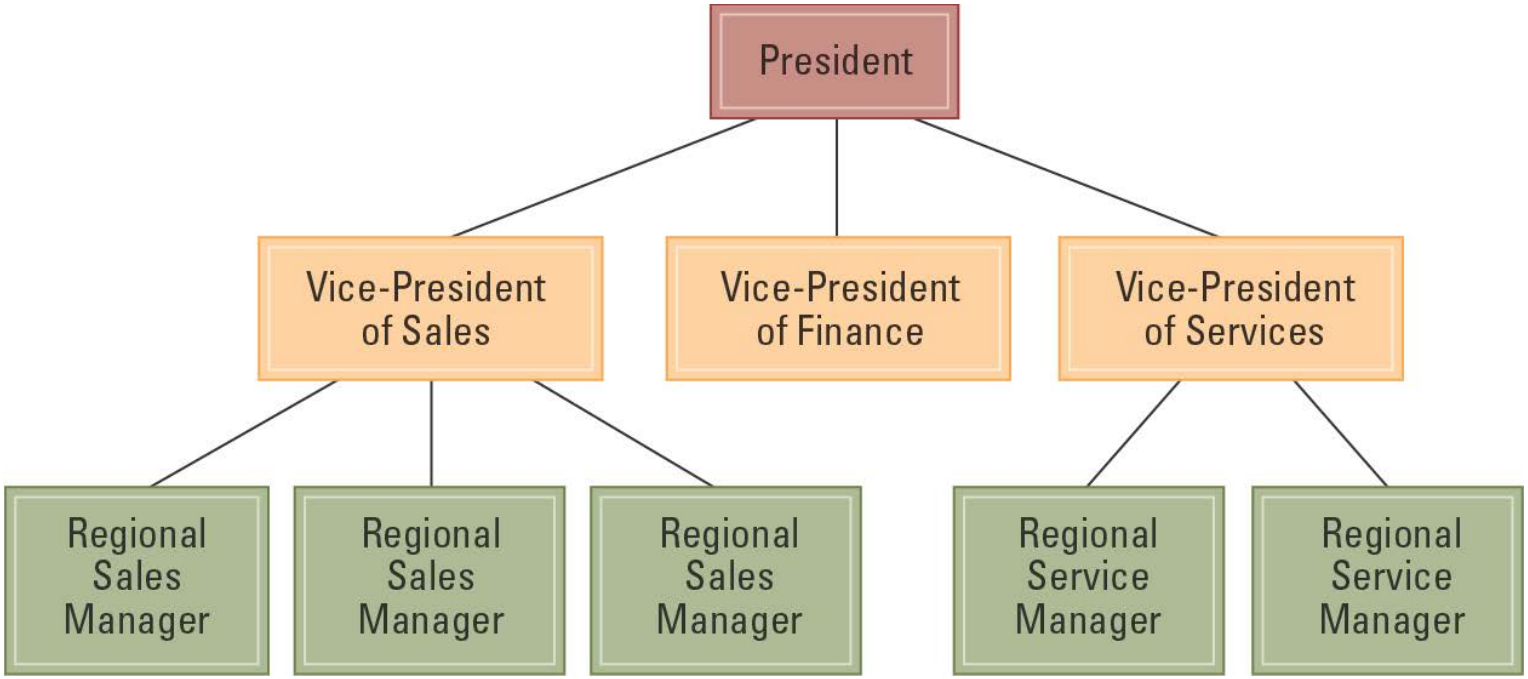
Terminology for Stacks

- **Stack:** A list in which entries are removed and inserted only at the head.
- **LIFO:** Last-in-first-out.
- **Top:** The head of list (stack).
- **Bottom** or **base:** The tail of list (stack).
- **Pop:** To remove the entry at the top.
- **Push:** To insert an entry at the top.

Terminology for Queues

- **Queue:** A list in which entries are removed at the head and are inserted at the tail.
- **FIFO:** First-in-first-out.

An Example of an Organization Chart

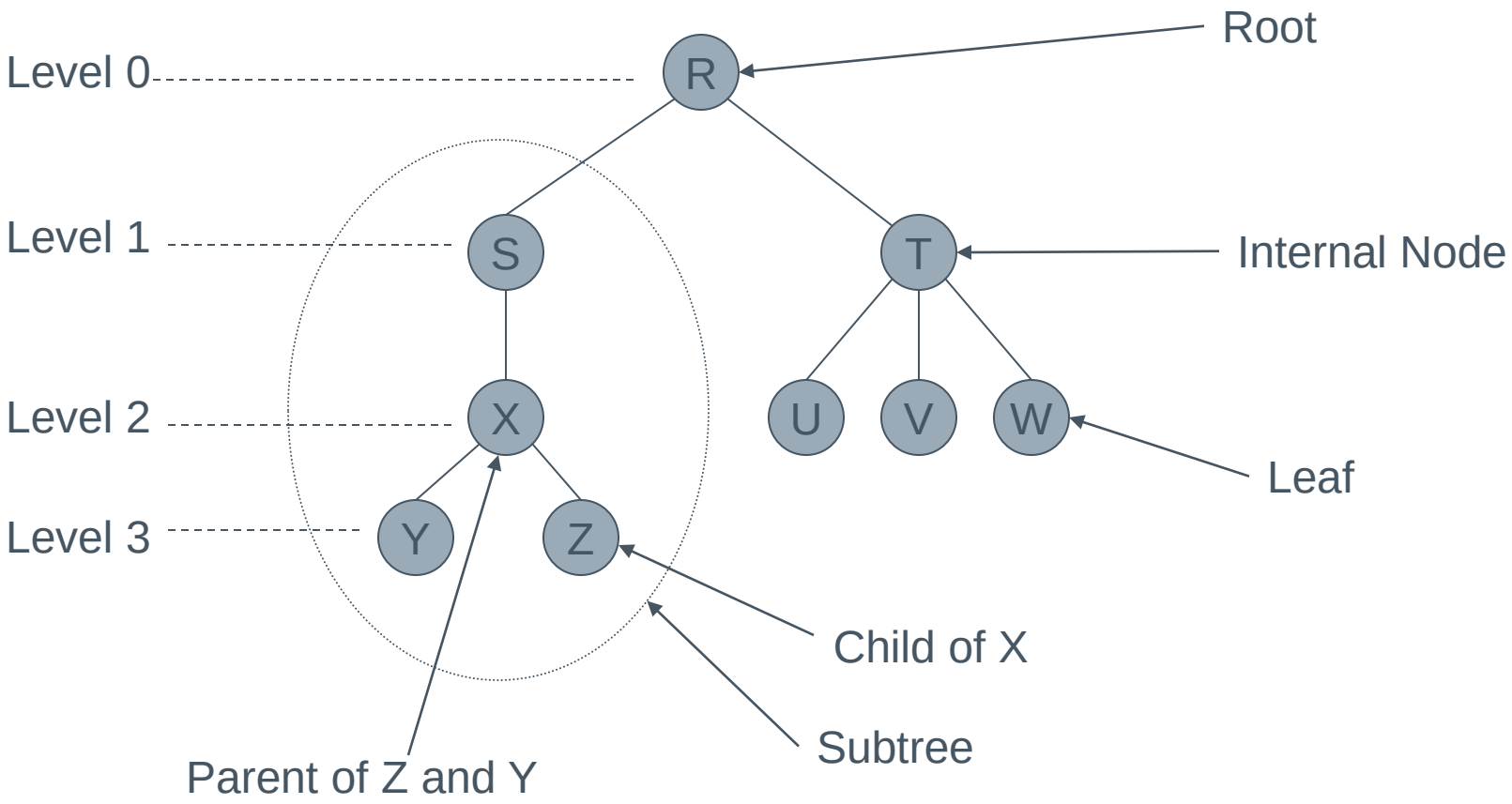


Terminology for a Tree

- **Tree:** A collection of data whose entries have a hierarchical organization.
- **Node:** An entry in a tree.
- **Root node:** The node at the top.
- **Terminal** or **leaf node:** A node at the bottom.
- **Parent:** The node immediately above a specified node
- **Child:** A node immediately below a specified node
- **Ancestor:** Parent, parent of parent, etc.
- **Descendent:** Child, child of child, etc.
- **Siblings:** Nodes sharing a common parent.

Tree Anatomy

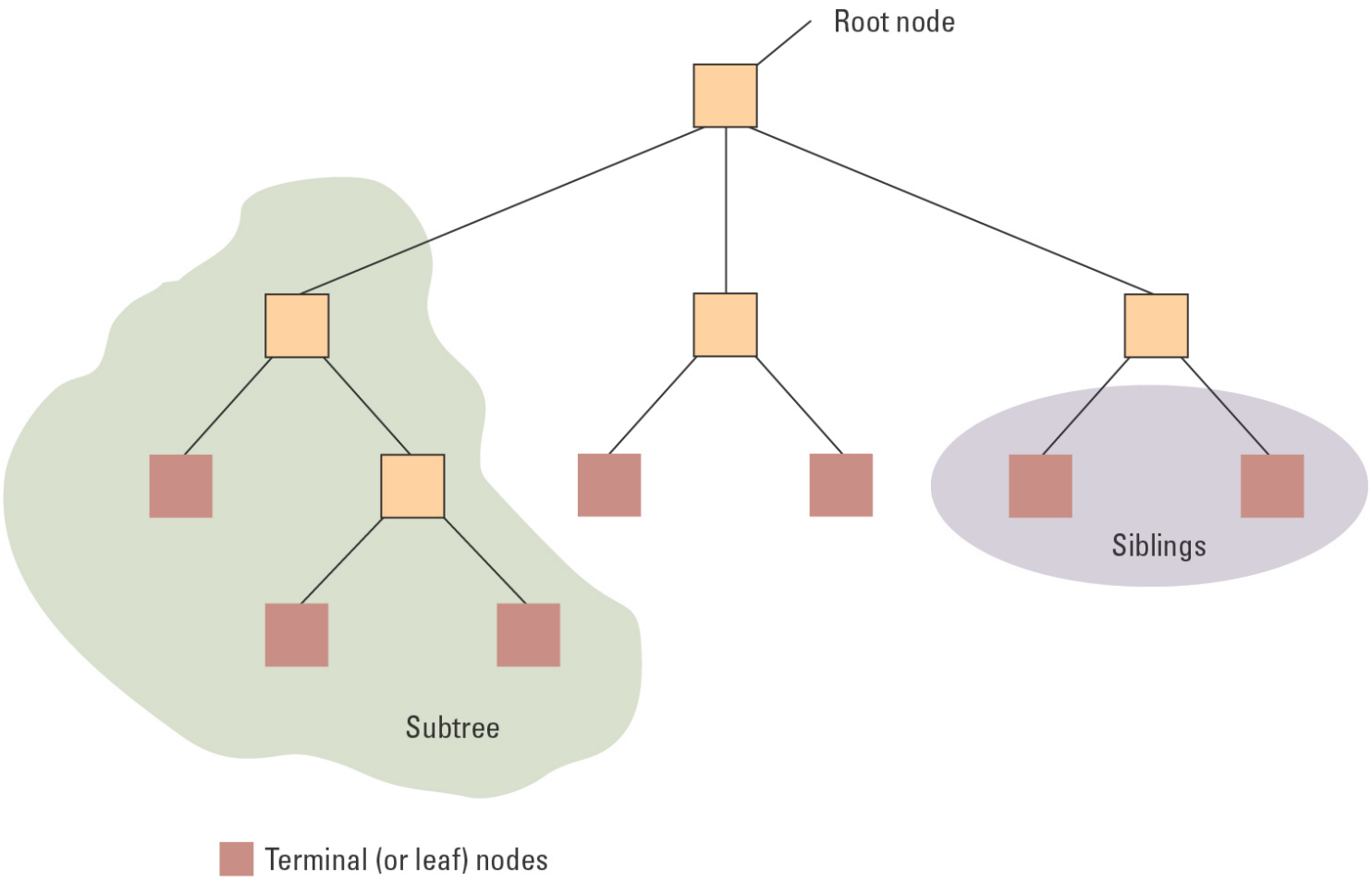
The children of a node are, themselves, trees, called subtrees.



Terminology for a Tree (continued)

- **Binary tree:** A tree in which every node has at most two children.
- **Depth:** The number of nodes in longest path from root to leaf.

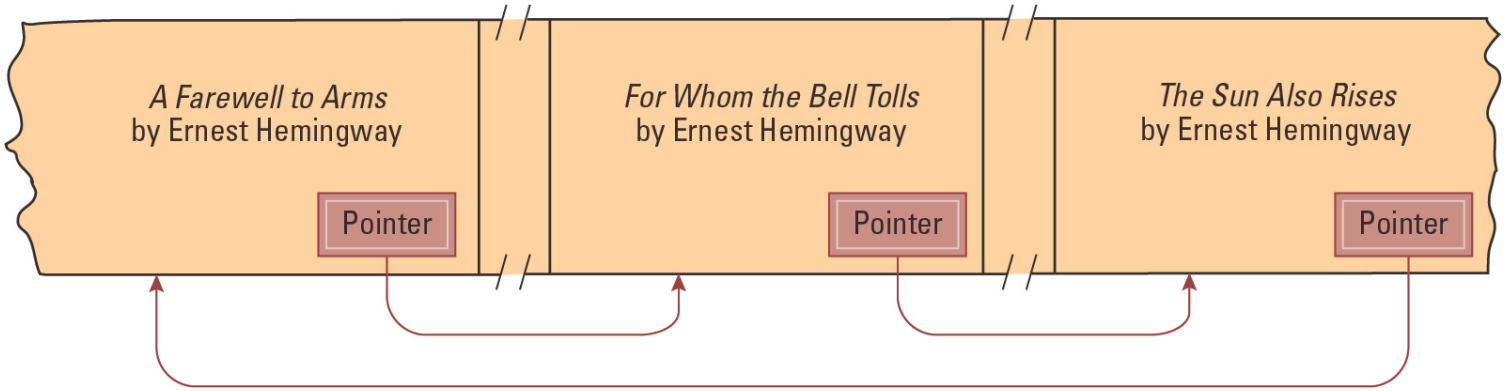
Tree Terminology



Related Concepts

- Abstraction
  - Shield *users* (application software) from details of actual data storage.
- Static vs. Dynamic Structure
  - Does the shape and size change over time?
- **Pointer**
  - A storage area that encodes an address where data is stored
  - Later used to access the data.

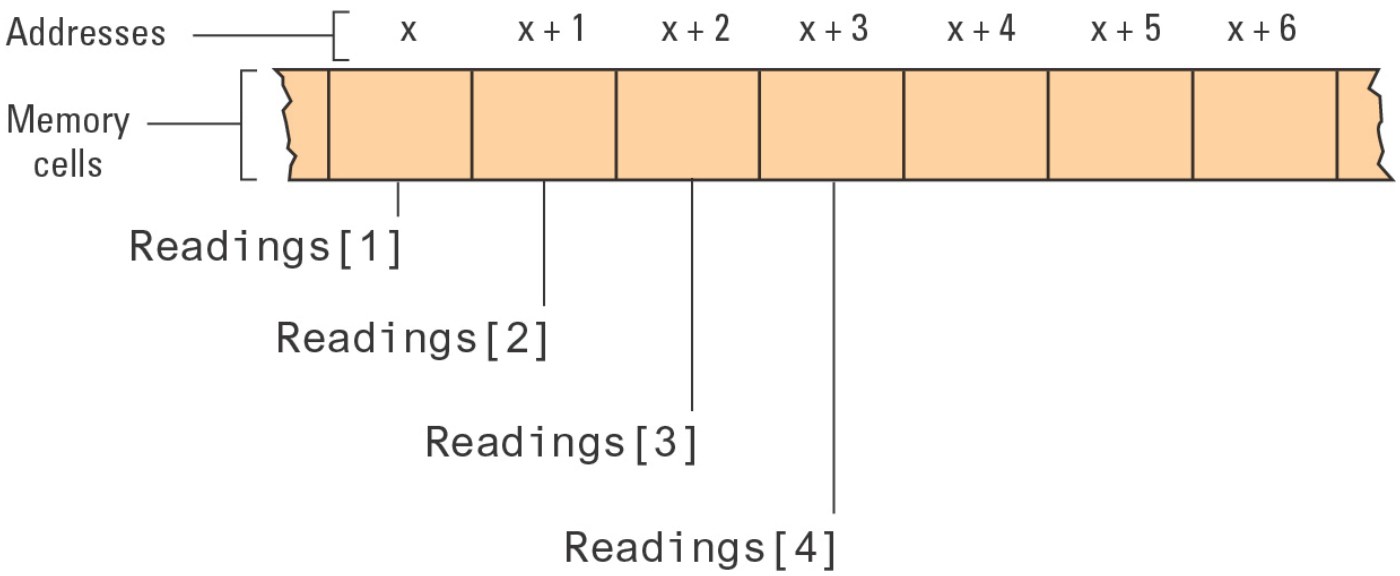
Novels Arranged by Title but Linked According to Authorship



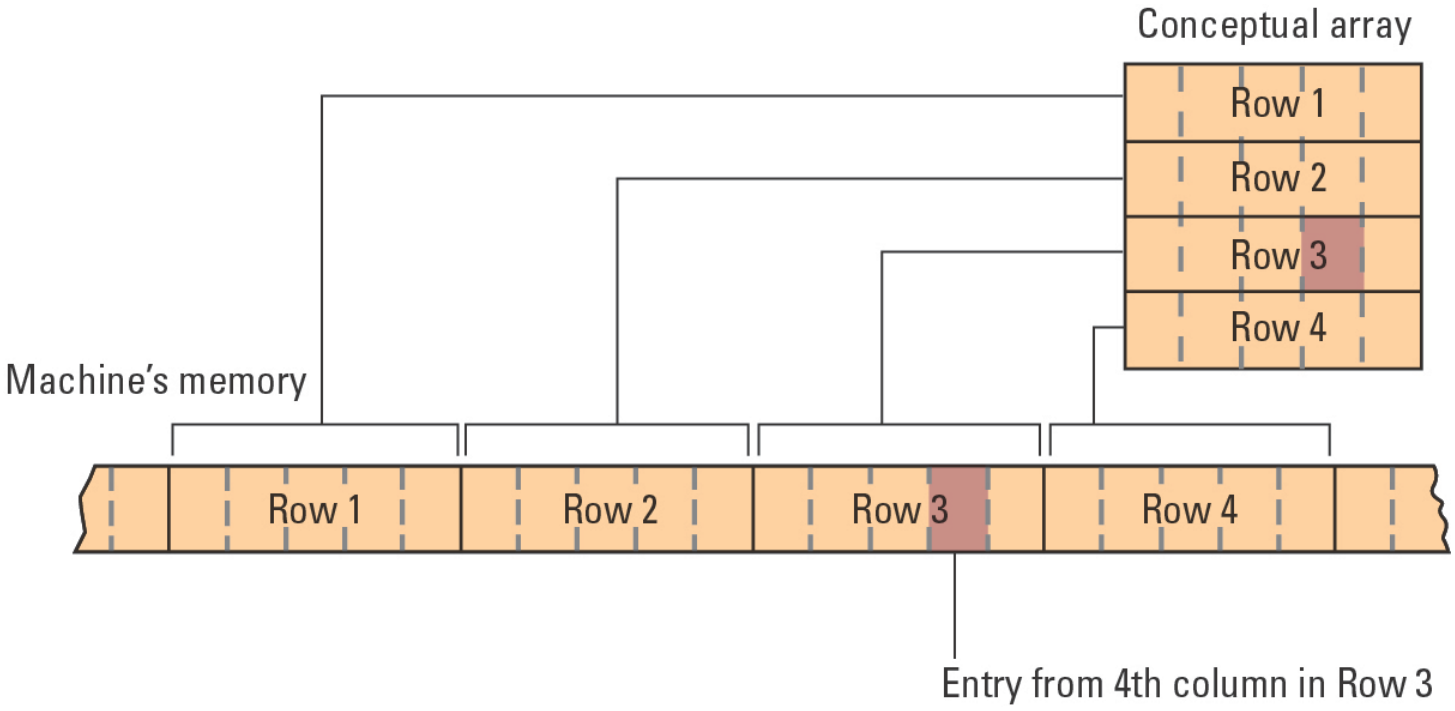
Storing Arrays

- Memory address of a particular cell can be computed.
- **Row-major order** versus **column major order**.
- Address polynomial.

The Array of Temperature Readings Stored in Memory Starting at Address x



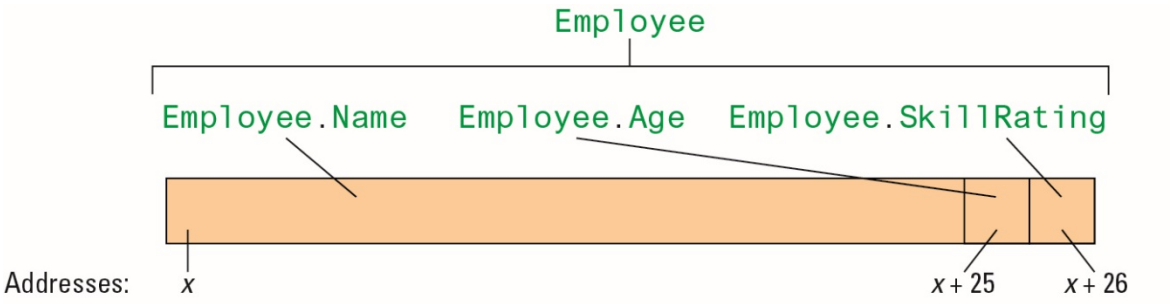
A Two-dimensional Array with Four Rows and Five Columns Stored in Row Major Order



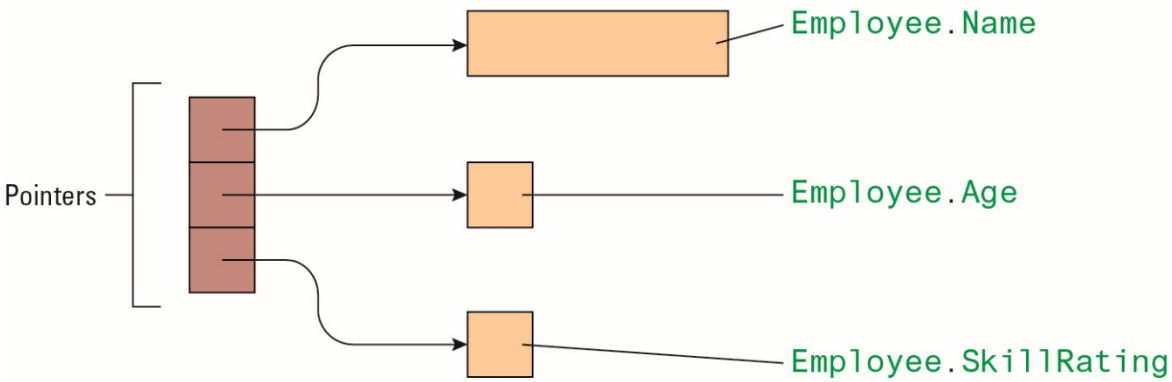
Storing Aggregates

- Fields can be stored one after the other in a contiguous block:
  - Memory cell address of each field can be computed.
- Fields can be stored in separate locations identified by pointers.

Storing the Aggregate Type `Employee`



a. Aggregate stored in a contiguous block

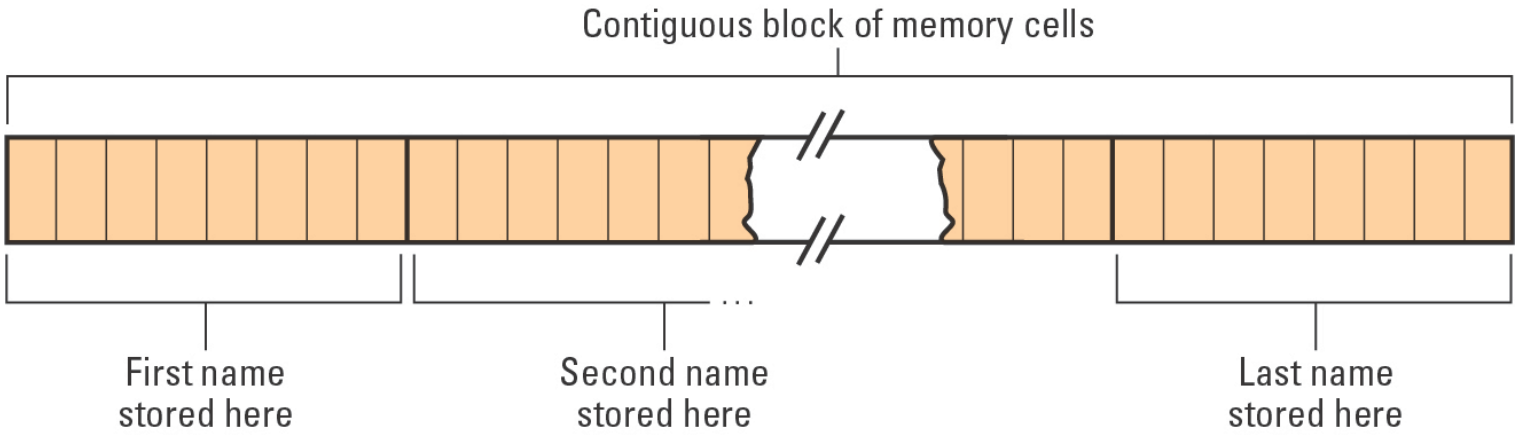


b. Aggregate fields stored in separate locations

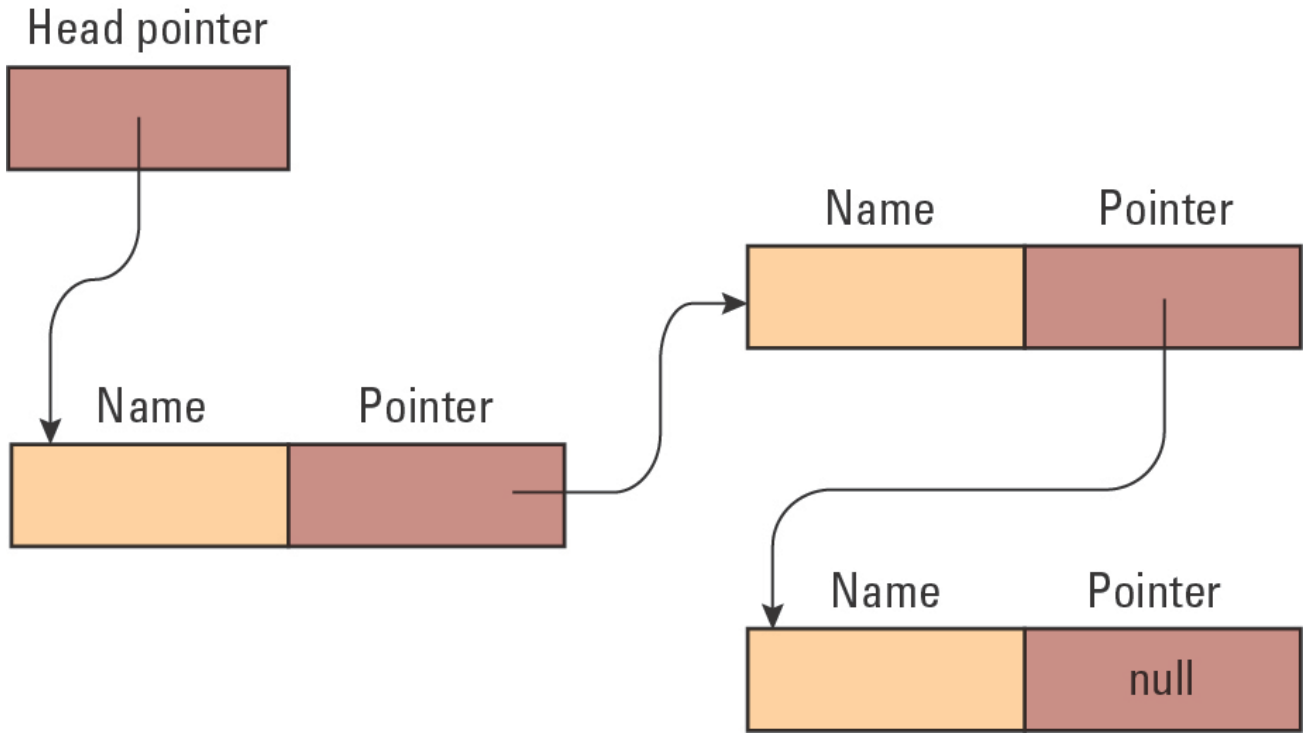
Storing Lists

- **Contiguous list:** List in which entries are stored in an array.
- **Linked list:** List in which entries are linked by pointers.
  - **Head pointer:** Pointer to first entry in list.
  - **null:** A “non-pointer” value used to indicate end of list .

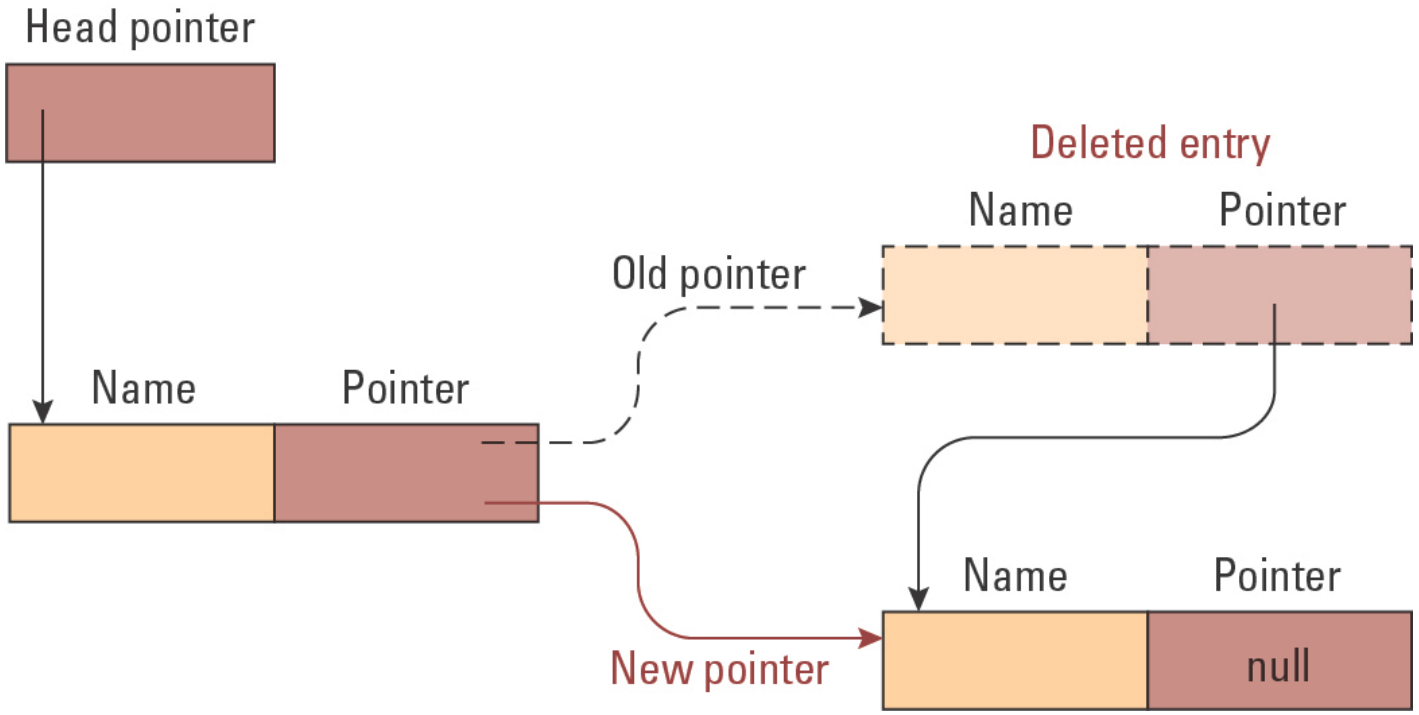
Names Stored in Memory as a Contiguous List



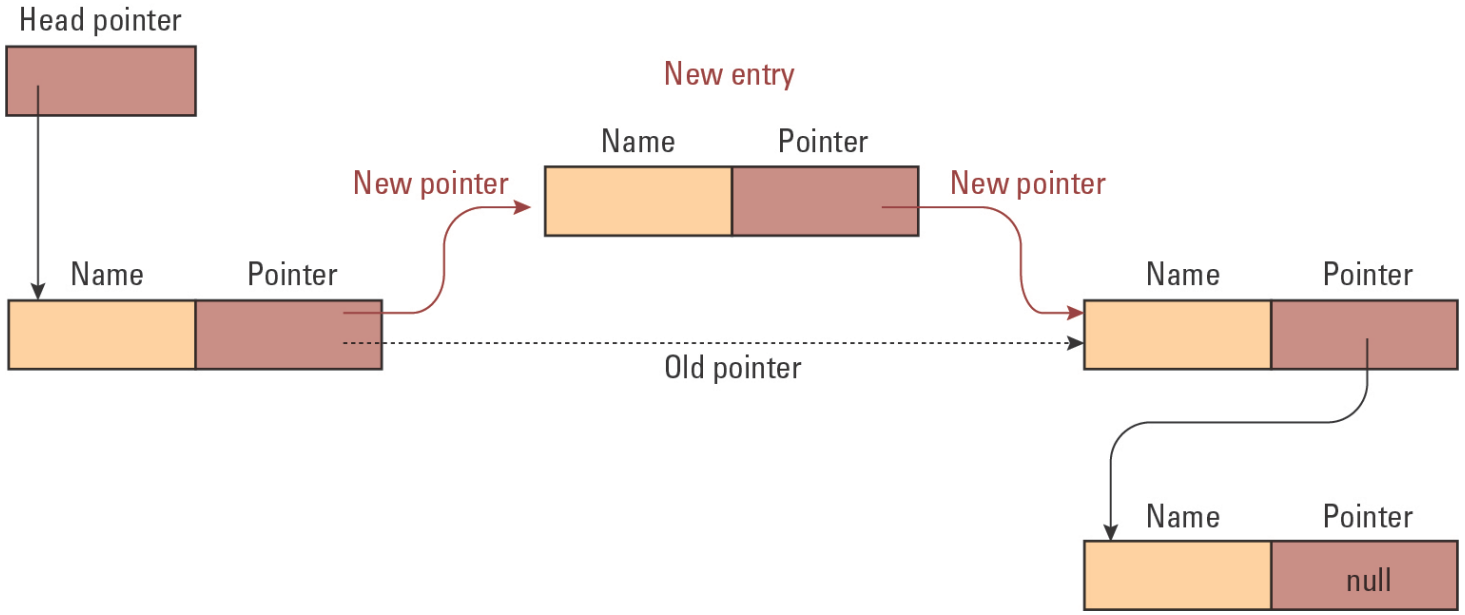
The Structure of a Linked List



Deleting an Entry From a Linked List



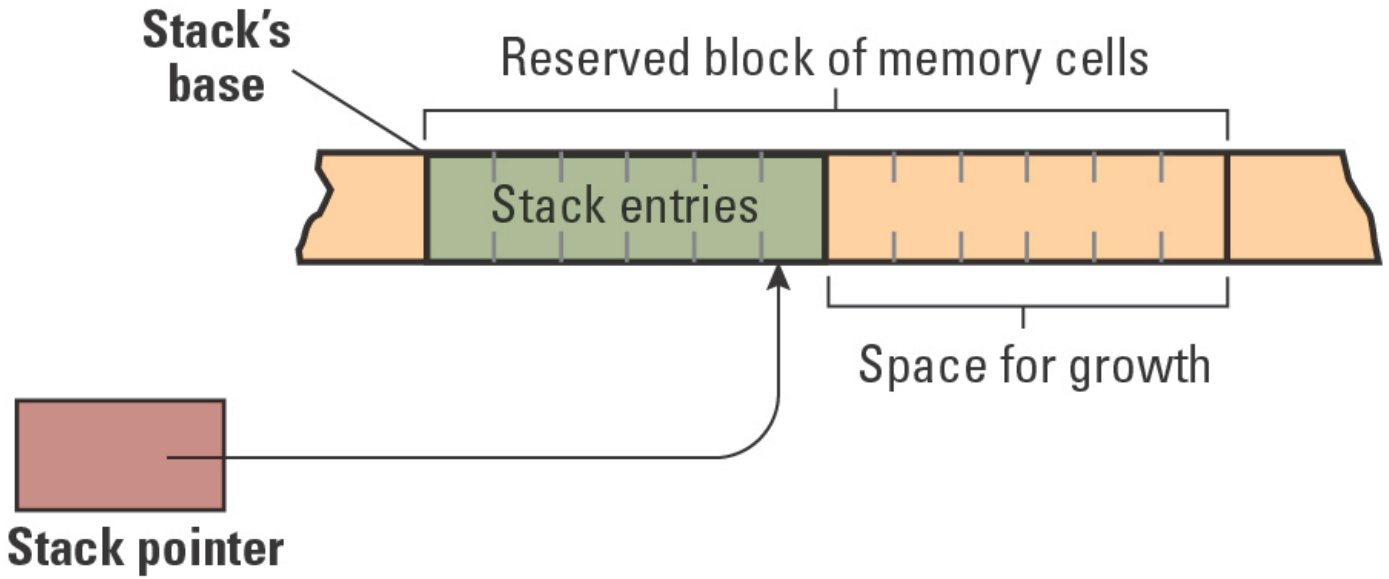
Inserting an Entry Into a Linked List



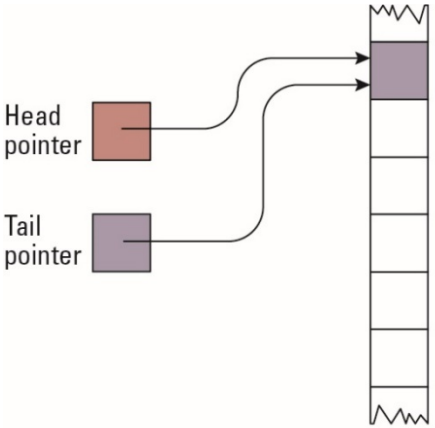
Storing Stacks and Queues

- Stacks usually stored as contiguous lists.
- Queues usually stored as **Circular Queues**
  - Stored in a contiguous block in which the first entry is considered to follow the last entry.
  - Prevents a queue from crawling out of its allotted storage space.

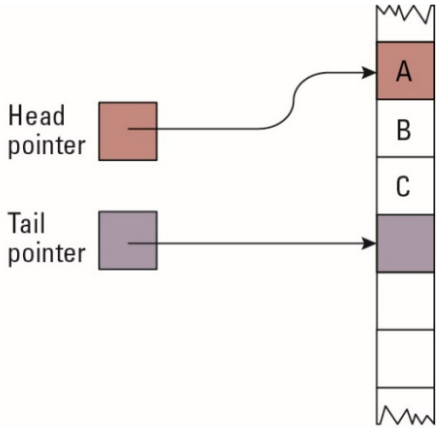
A Stack in Memory



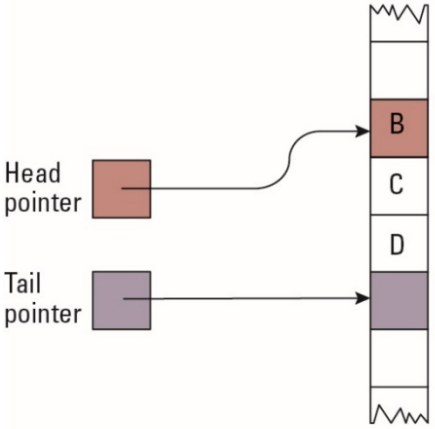
A Queue Implementation with Head and Tail Pointers



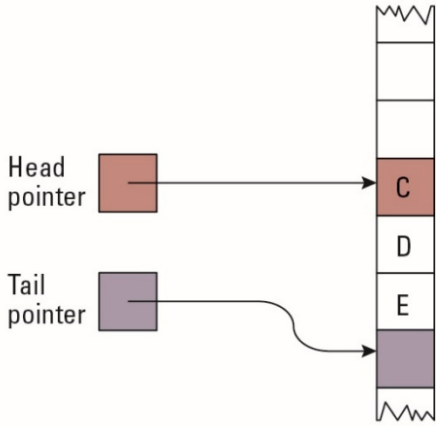
a. Empty queue



b. After inserting entries A, B, and C

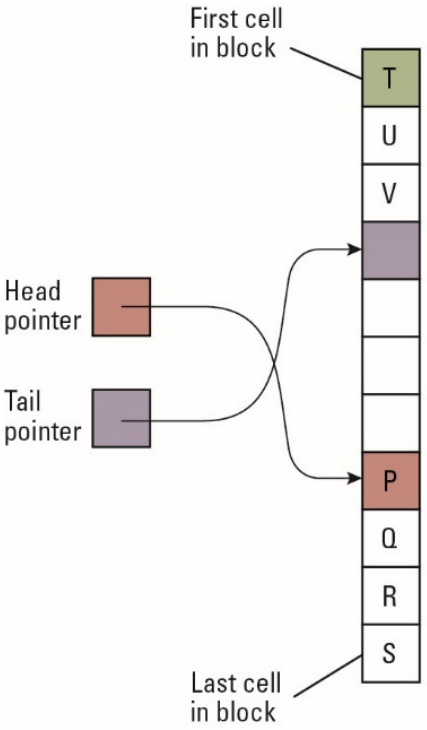


c. After removing A and inserting D

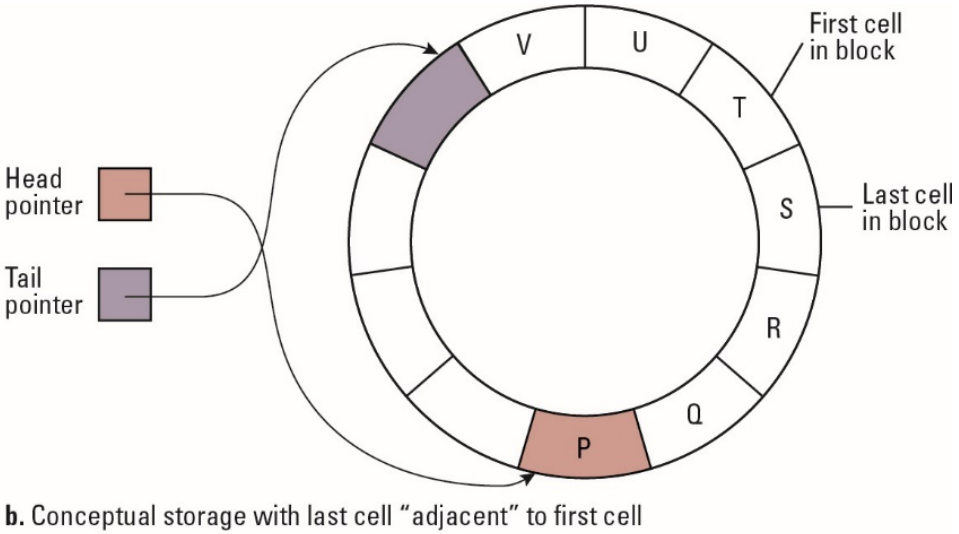


d. After removing B and inserting E

A Circular Queue Containing the Letters P Through V



a. Queue as actually stored



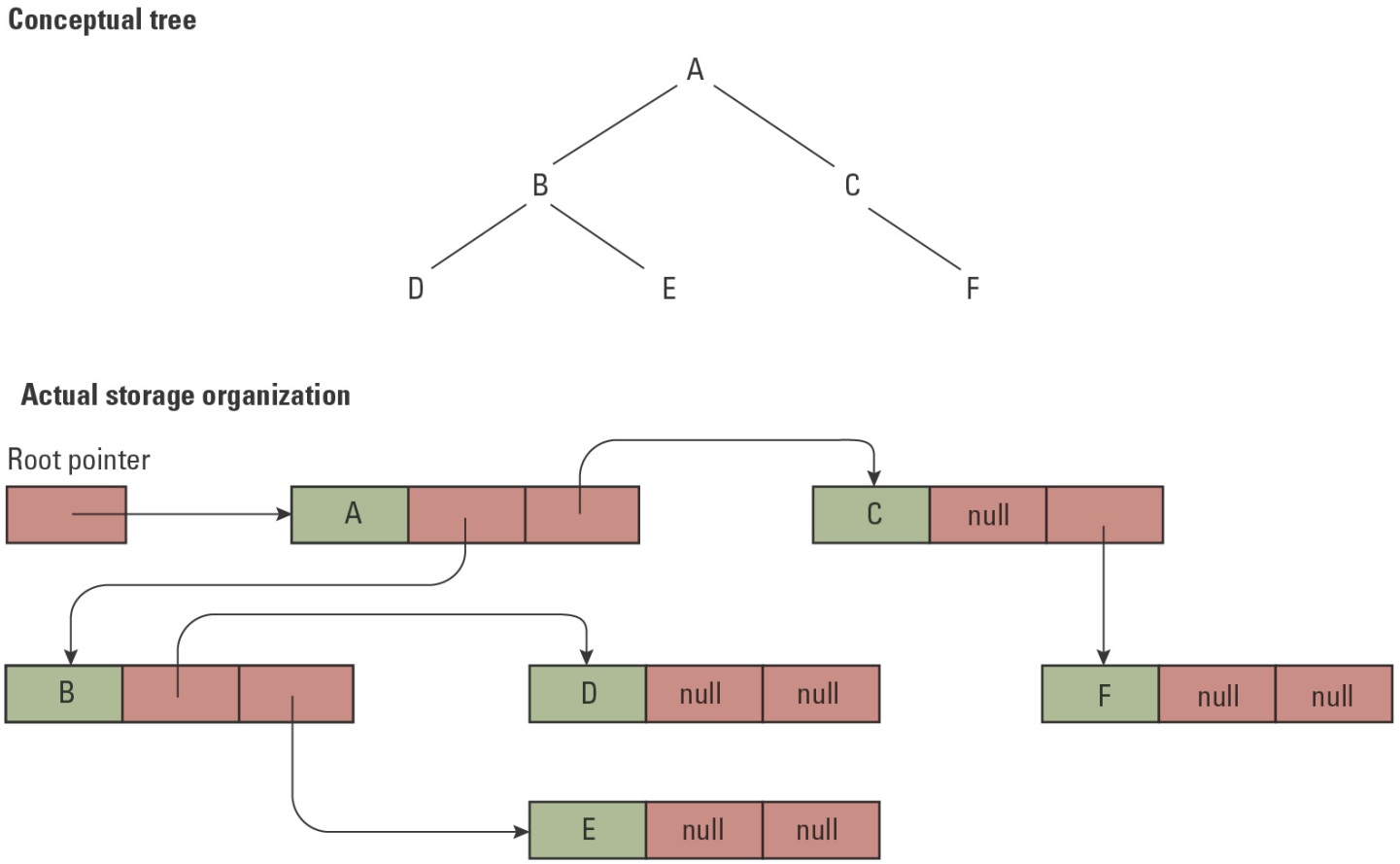
b. Conceptual storage with last cell "adjacent" to first cell

Storing Binary Trees

- Linked structure.
  - Each node = data cells + two child pointers.
  - Accessed via a pointer to root node.
- Contiguous array structure.
  - `A[1]` = root node.
  - `A[2],A[3]` = children of `A[1]`.
    - `A[4],A[5]` = children of `A[2]` .
    - `A[6],A[7]` = children of `A[3]` .

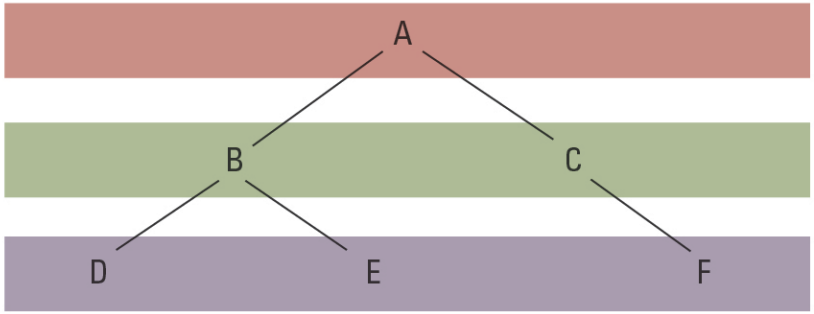
The Structure of a Node in a Binary Tree

The Conceptual and Actual Organization of a Binary Tree Using a Linked Storage System

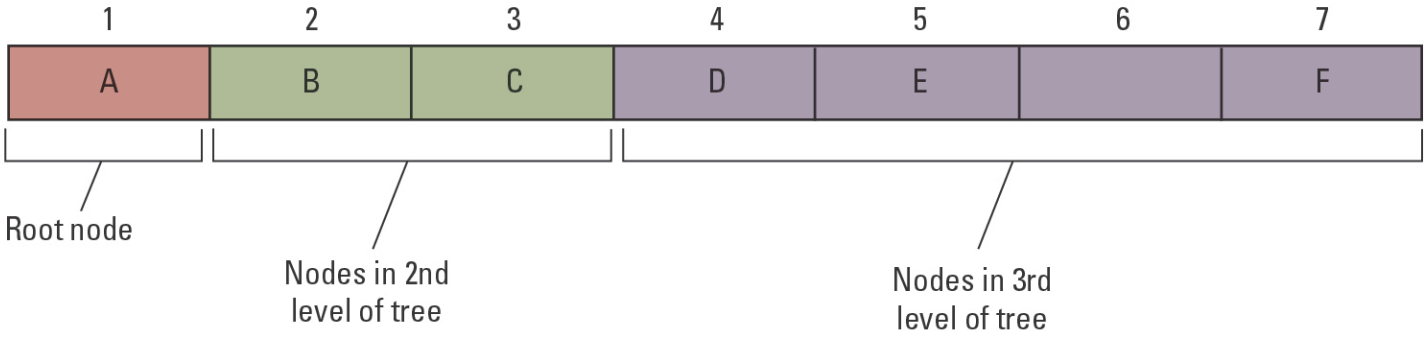


A Tree Stored Without Pointers

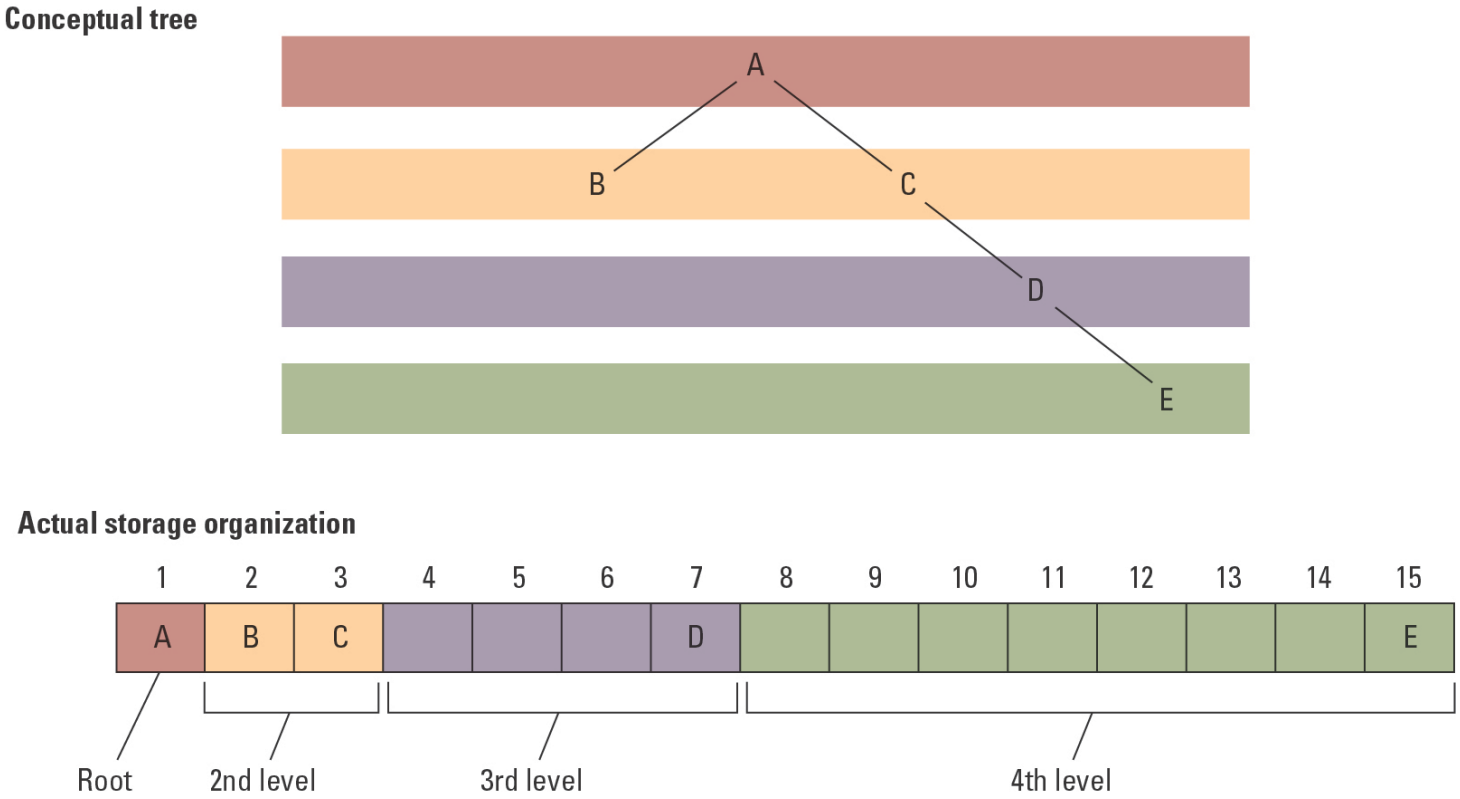
Conceptual tree



Actual storage organization



A Sparse, Unbalanced Tree Shown in its Conceptual Form and as it Would be Stored Without Pointers



Manipulating Data Structures

- Ideally, a data structure should be manipulated solely by pre-defined functions.
  - Example: A list typically has a function `insert` for inserting new entries.
  - The data structure along with these functions constitutes a complete abstract tool.

A Function for Printing a Linked List

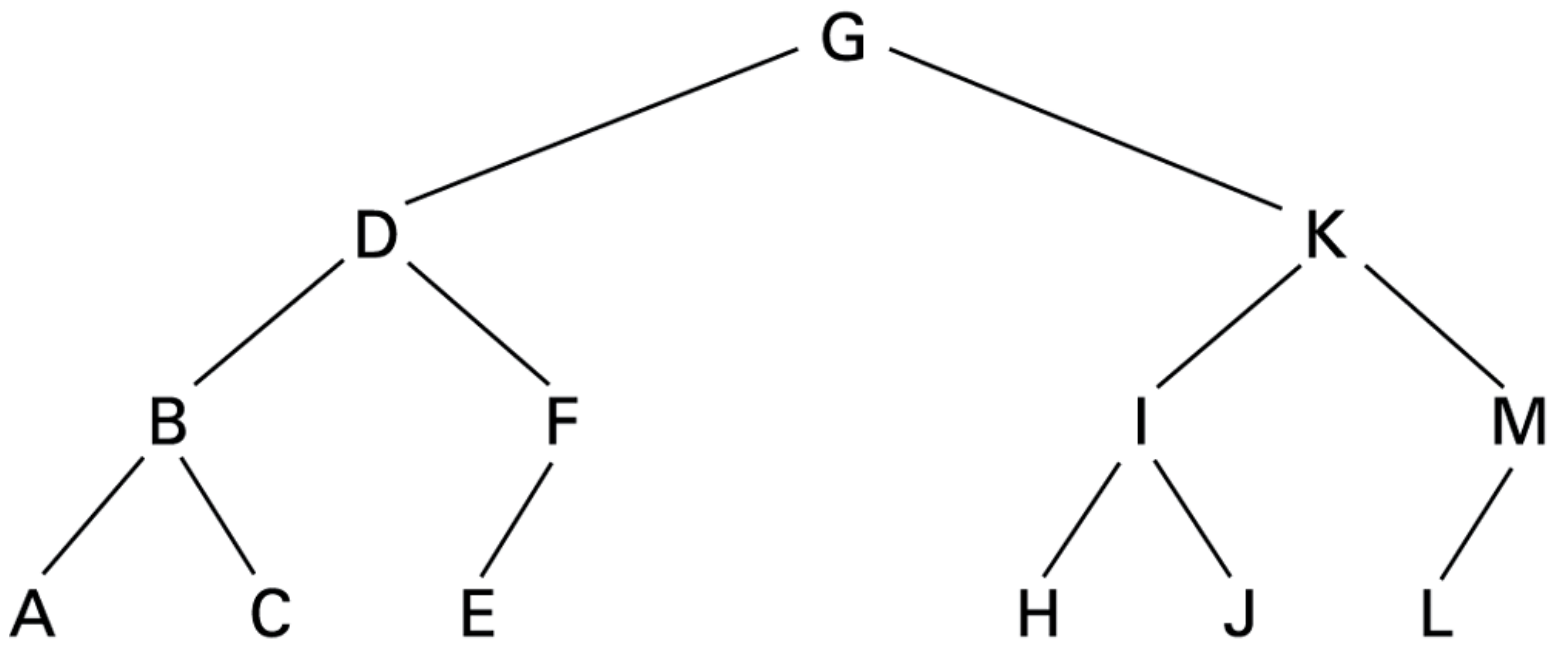
```
def PrintList (List):
    CurrentPointer = List.Head
    while (CurrentPointer is not None):
        print(CurrentPointer.Value)
        CurrentPointer = CurrentPointer.Next
```

Case Study

Problem:

Construct an abstract tool consisting of a list of names in alphabetical order along with the operations: search, print, and insert.

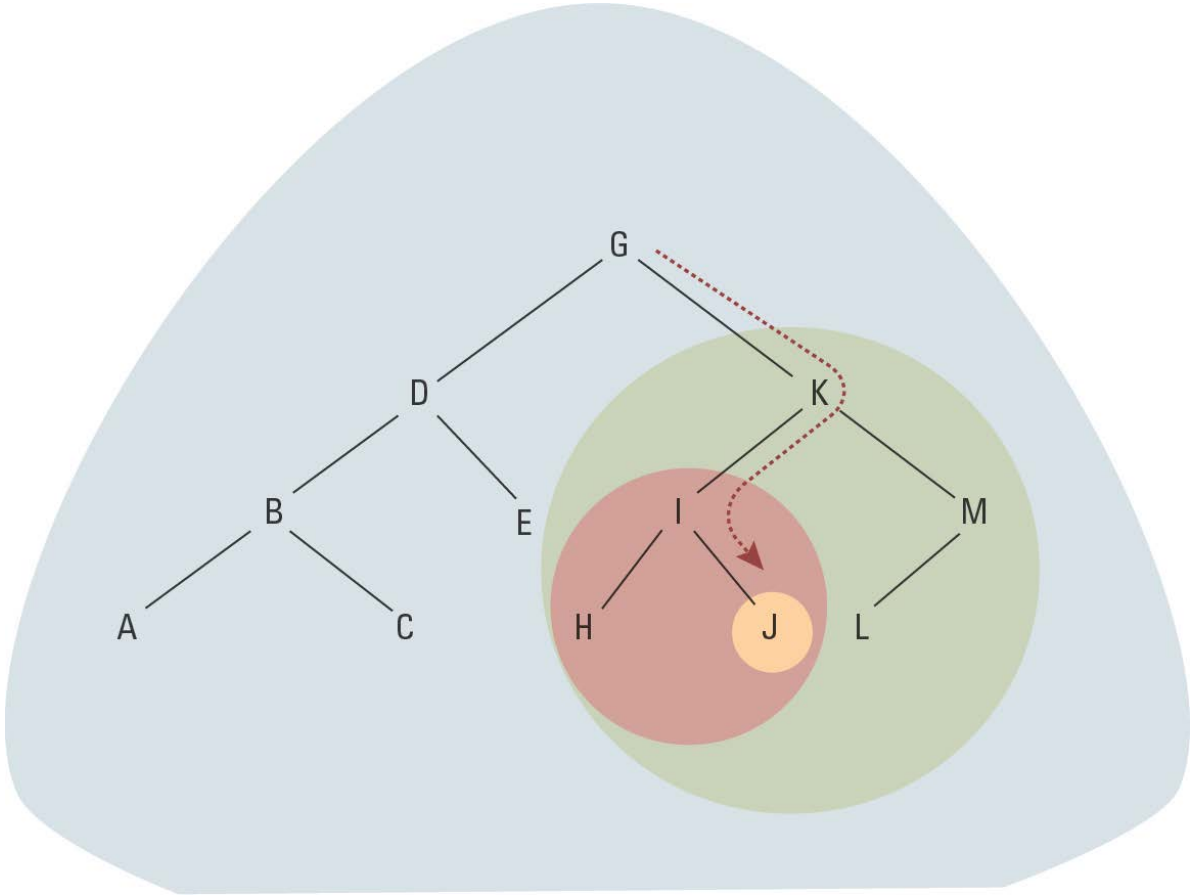
The Letters A Through M Arranged in an Ordered Tree



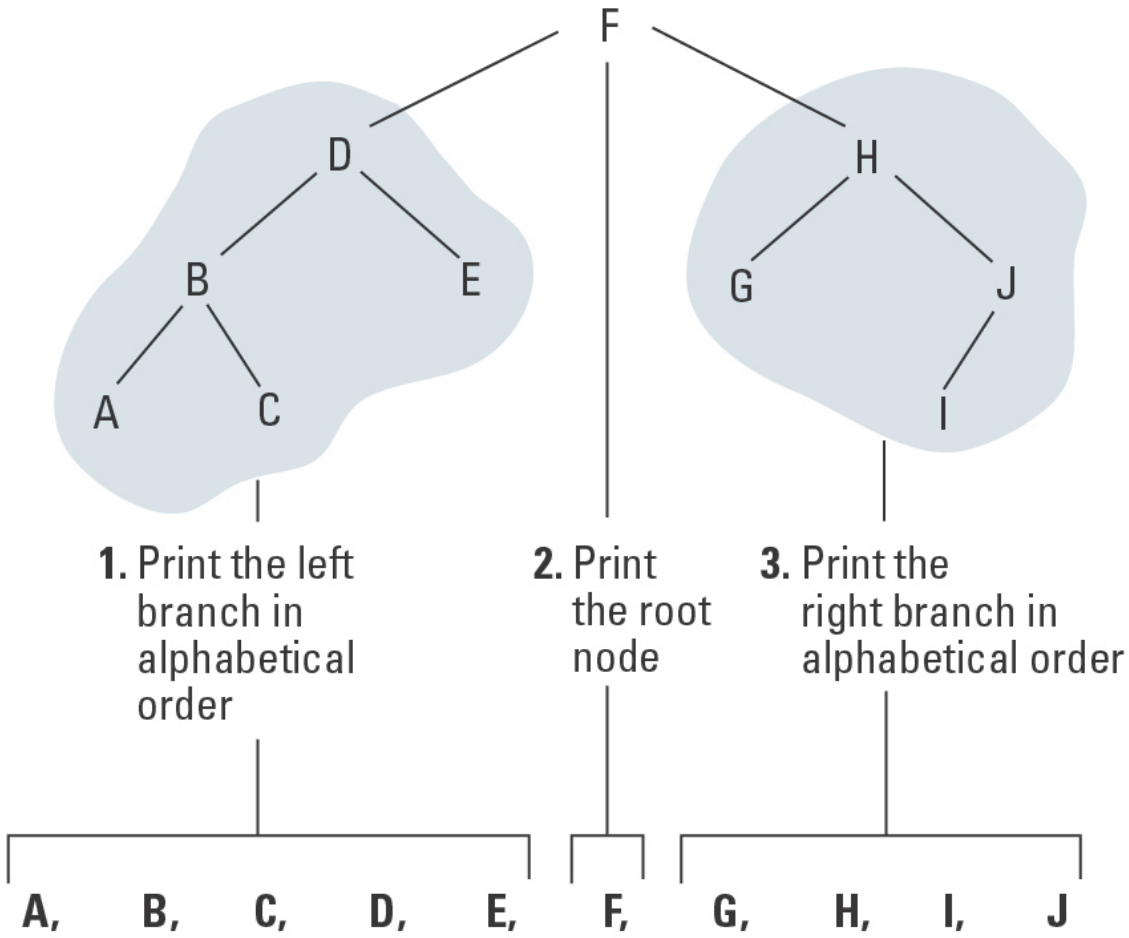
The Binary Search as it Would Appear if the List were Implemented as a Linked Binary Tree

```
def Search (Tree, TargetValue):
    if (Tree is None):
        return None        # Search failed
    elif (TargetValue == Tree.Value):
        return Tree        # Search succeeded
    elif (TargetValue < Tree.Value):
        # Continue search in left subtree
        return Search(Tree.Left, TargetValue)
    elif (TargetValue > Tree.Value):
        # Continue search in right subtree
        return Search(Tree.Right, TargetValue)
```

The Successively Smaller Trees Considered by the Function when Searching for the Letter **J**



Printing a Search Tree in Alphabetical Order

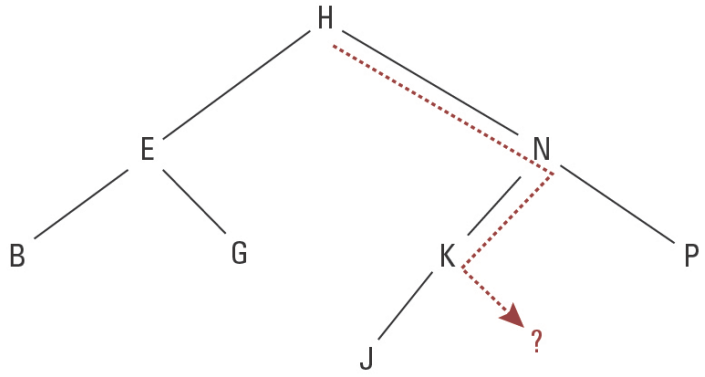


A Function for Printing the Data in a Binary Tree

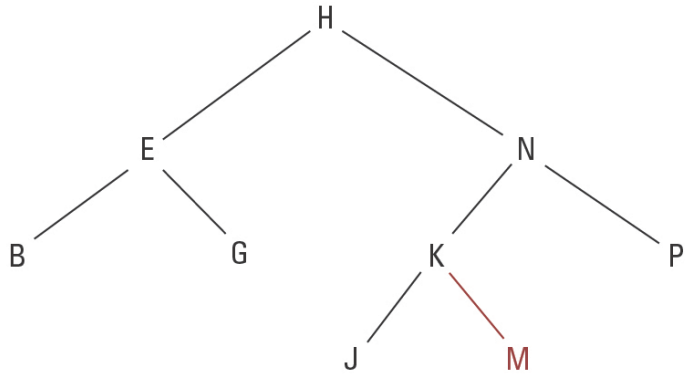
```
def PrintTree (Tree):
    if (Tree is not None):
        PrintTree(Tree.Left)
        print(Tree.Value)
        PrintTree(Tree.Right)
```

Inserting the Entry M into the List B, E, G, H, J, K, N, P Stored as a Tree

a. Search for the new entry until its absence is detected



b. This is the position in which the new entry should be attached



A Function for Inserting a New Entry in a List Stored as a Binary Tree

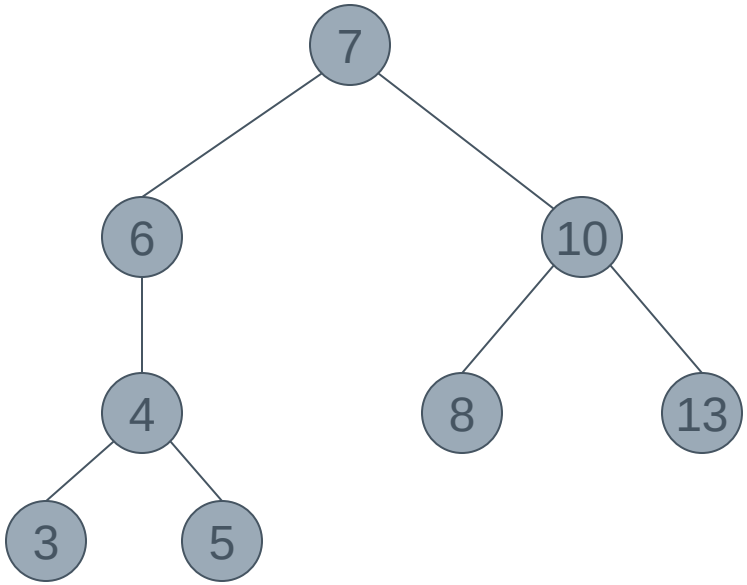
```
def Insert(Tree, NewValue):
    if (Tree is None):
        # Create a new leaf with NewValue
        Tree = TreeNode()
        Tree.Value = NewValue
    elif (NewValue < Tree.Value):
        # Insert NewValue into the left subtree
        Tree.Left = Insert(Tree.Left, NewValue)
    elif (NewValue > Tree.Value):
        # Insert NewValue into the right subtree
        Tree.Right = Insert(Tree.Right, NewValue)
    else:
        # Make no change
    return Tree
```

Tree Traversals

- One of the most common operations performed on trees, are a *tree traversals*
- A traversal starts at the root of the tree and visits every node in the tree exactly once
  - *visit means to process the data in the node*
- Traversals are either *depth-first* or *breadth-first*

Breadth First Traversals

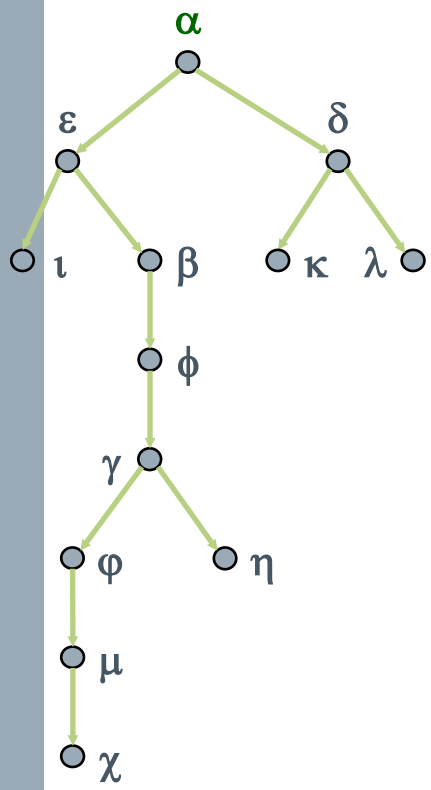
- All the nodes in one level are visited
- Followed by the nodes the at next level
- Beginning at the root
- For the sample tree
  - 7, 6, 10, 4, 8, 13, 3, 5



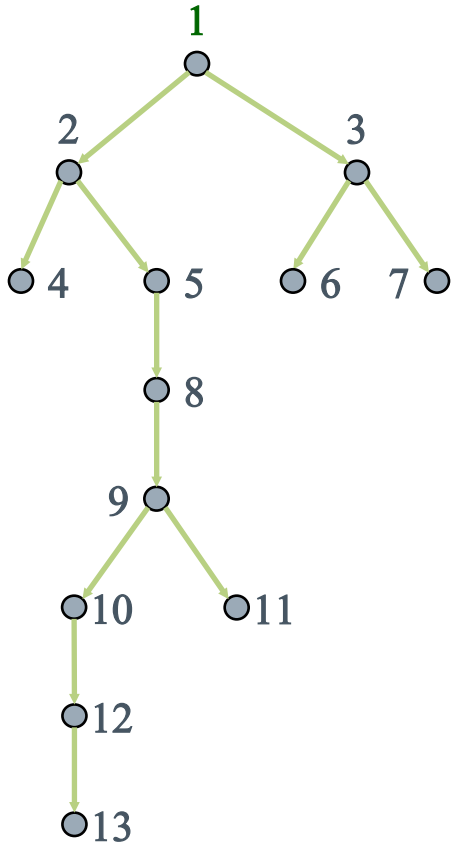
Breadth first tree traversal with a queue

- Enqueue root
- While queue is not empty
  - Dequeue a vertex and write it to the output list
  - Enqueue its children left-to-right

Breadth first tree traversal with a queue



Step	*Output*	*Queue*
0		α
1	α	ε,δ
2	ε	δ,ι,β
3	δ	ι,β,κ,λ
4	ι	β,κ,λ
5	β	κ,λ,ϕ
6	κ	λ,ϕ
7	λ	ϕ
8	ϕ	γ
9	γ	φ,η
10	φ	η,μ
11	η	μ,χ
12	μ	χ
13	χ	

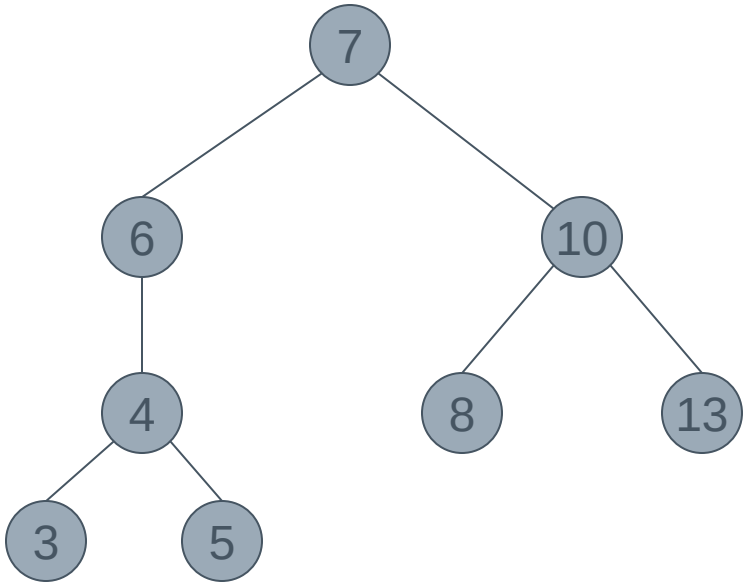


Depth-First Traversals

- There are 8 different depth-first traversals
  - VLR (pre-order traversal)
  - VRL
  - LVR (in-order traversal)
  - RVL
  - RLV
  - LRV (post-order traversal)

Pre-order Traversal: VLR

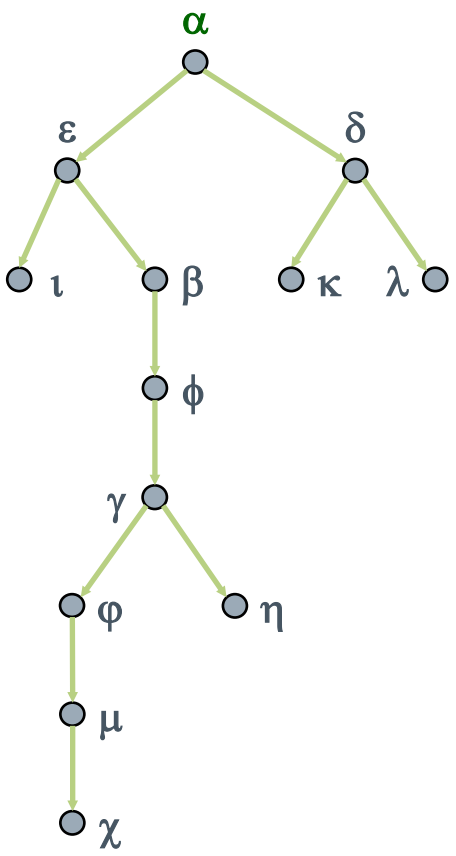
- Visit the node
- Do a pre-order traversal of the left subtree
- Finish with a pre-order traversal of the right subtree
- For the sample tree
  - 7, 6, 4, 3, 5, 10, 8, 13



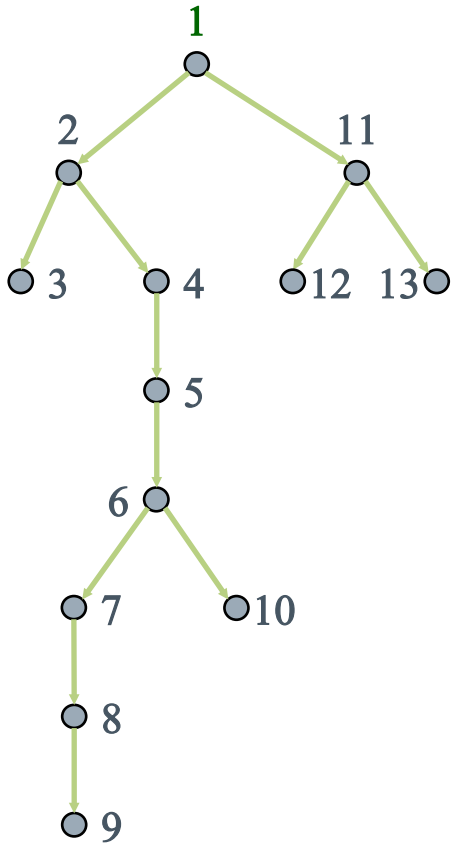
Pre-order tree traversal with a stack

- Push root onto the stack
- While stack is not empty
  - Pop a vertex off stack, and write it to the output list
  - Push its children right-to-left onto stack

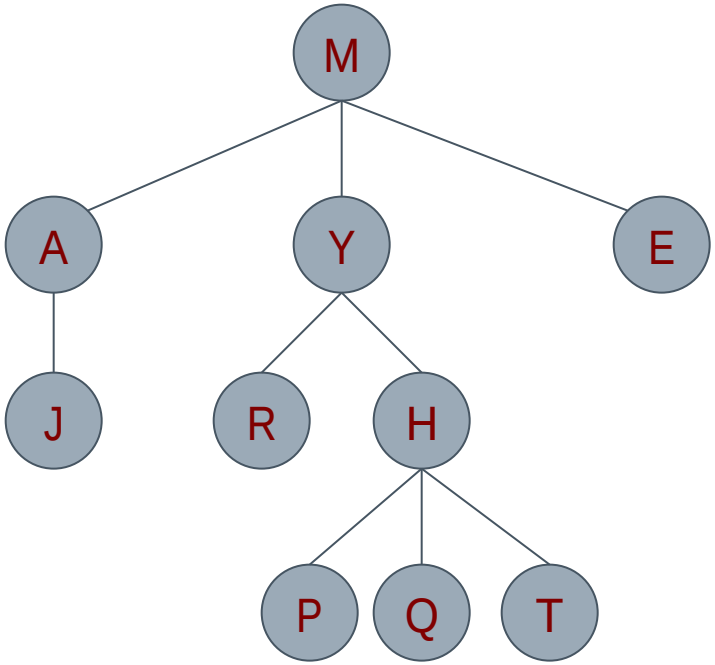
Pre-order tree traversal with a stack



Step	Output	Stack
0		α
1	α	δ,ε
2	ε	δ,β,ι
3	ι	δ,β
4	β	δ,ϕ
5	ϕ	δ,γ
6	γ	δ,η,φ
7	φ	δ,η,μ
8	μ	δ,η,χ
9	χ	δ,η
10	η	δ
11	δ	λ,κ
12	κ	λ
13	λ	

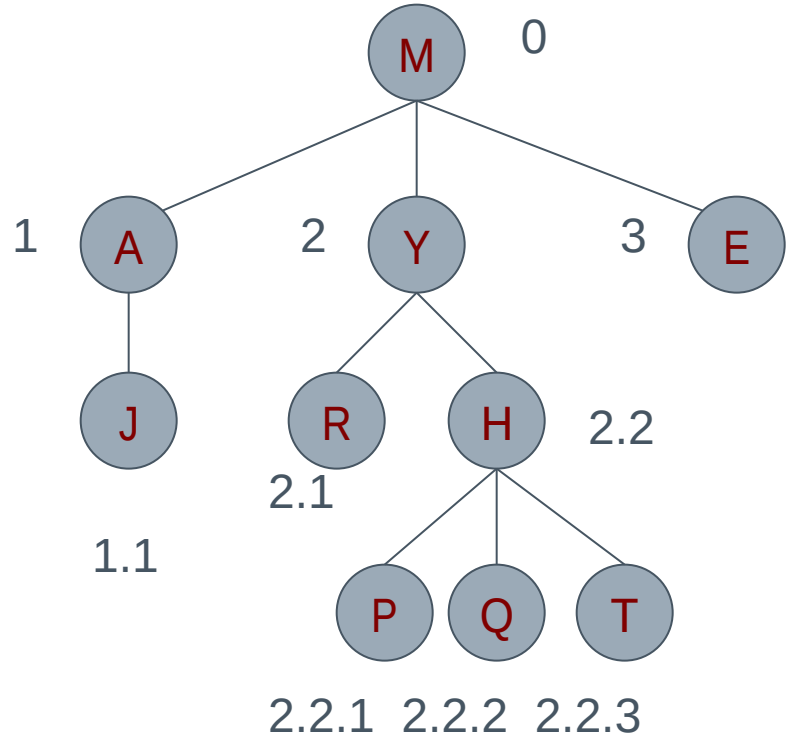


Example



M A J Y R H P Q T E

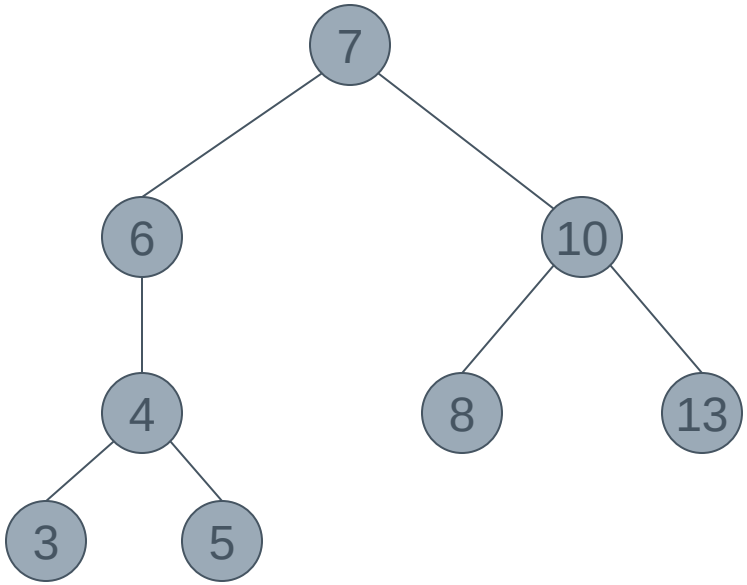
Ordering of the preorder traversal is the same a the **Universal Address System** with lexicographic ordering.



M A J Y R H P Q T E

In-order Traversal: LVR

- Do an in-order traversal of the left subtree
- Visit the node
- Finish with an in-order traversal of the right subtree
- For the sample tree
  - 3, 4, 5, 6, 7, 8, 10, 13

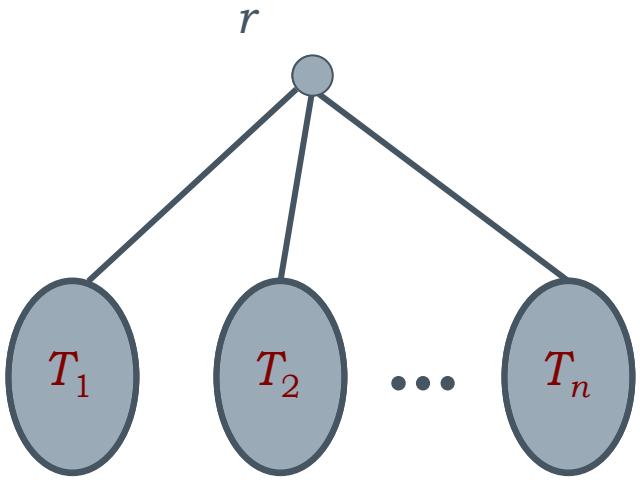


Inorder Traversal

Step 1: Visit T_1 in inorder

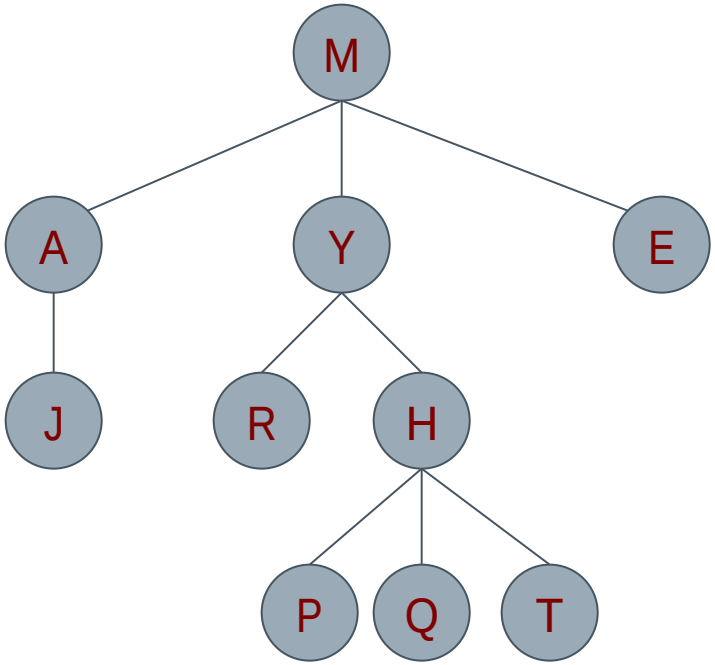
Step 2: Visit r

Step 3: Visit T_2 in inorder



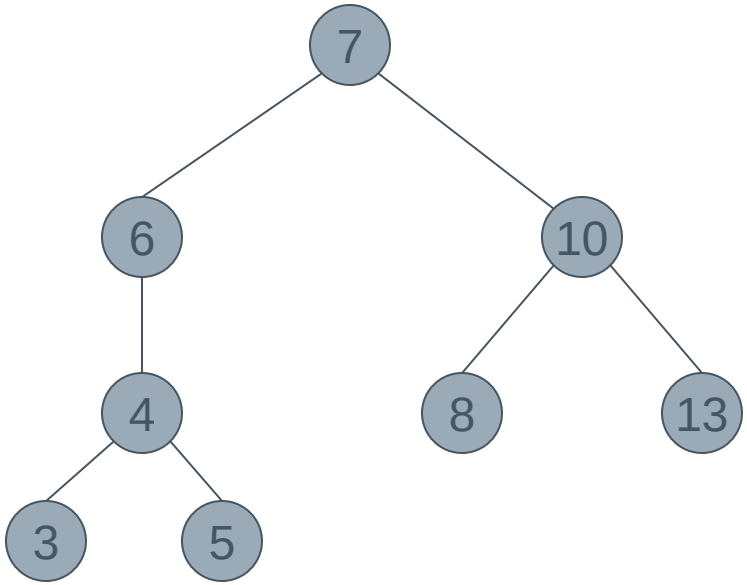
Step n+1: Visit T_n in inorder

Example



Post-order Traversal: LRV

- Do a post-order traversal of the left subtree
- Followed by a post-order traversal of the right subtree
- Visit the node
- For the sample tree
  - 3, 5, 4, 6, 8, 13, 10, 7



Postorder Traversal

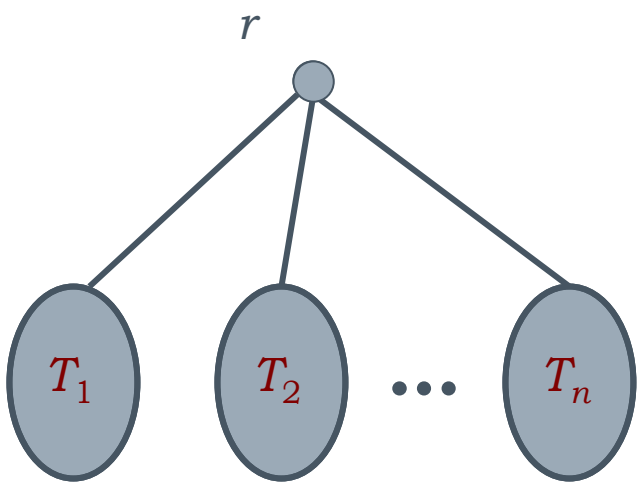
Step 1: Visit T_1 in postorder

Step 2: Visit T_2 in postorder

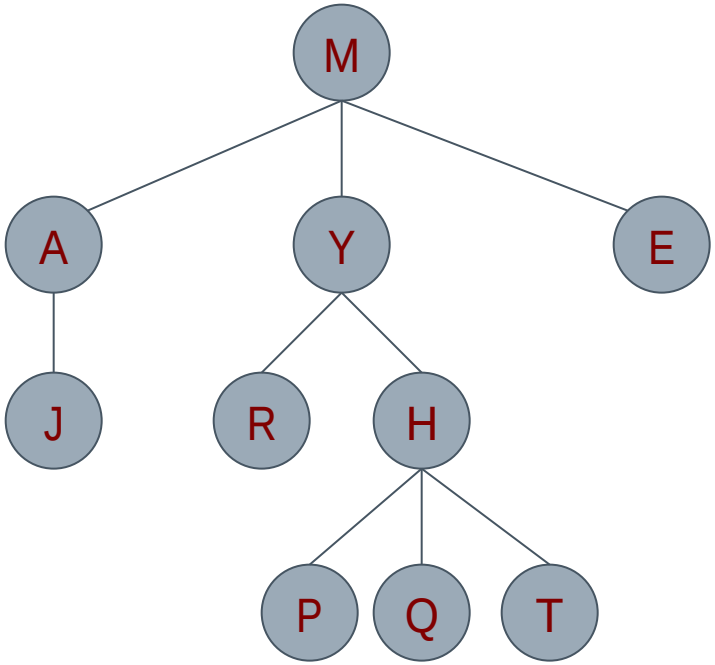
⋮

Step n: Visit T_n in postorder

Step n+1: Visit r



Example

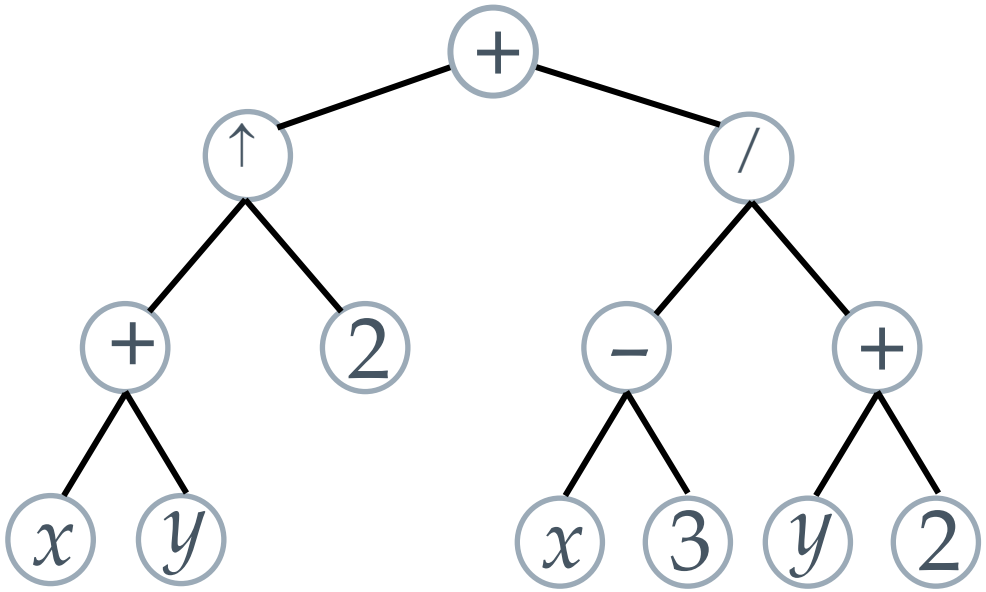


Representing Arithmetic Expressions

- Complicated arithmetic expressions can be represented by an ordered rooted tree
  - Internal vertices represent operators
  - Leaves represent operands
- Build the tree bottom-up
  - Construct smaller subtrees
  - Incorporate the smaller subtrees as part of larger subtrees

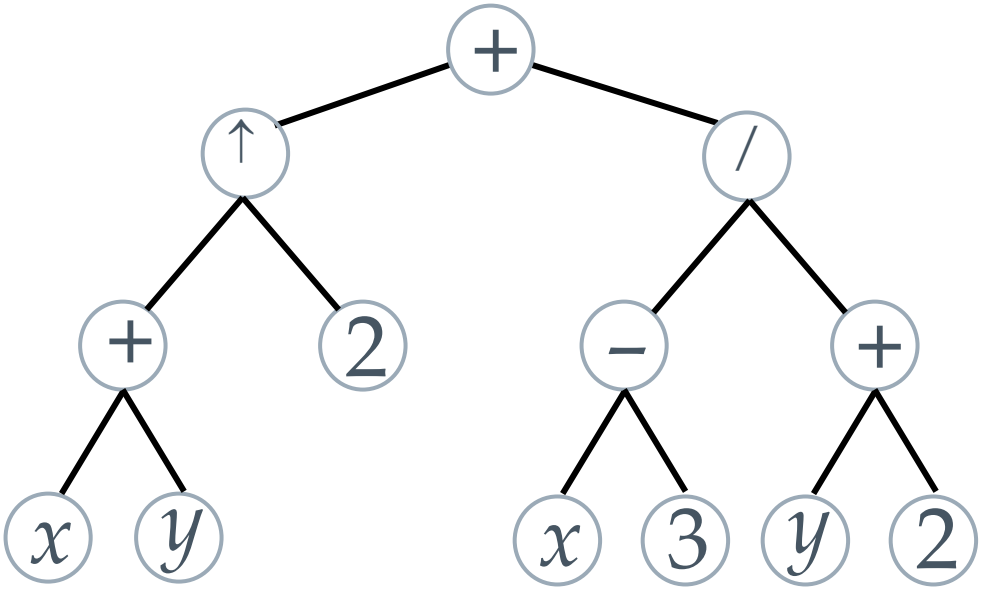
Example

› (x+y)2 + (x-3)/(y+2)



Infix Notation

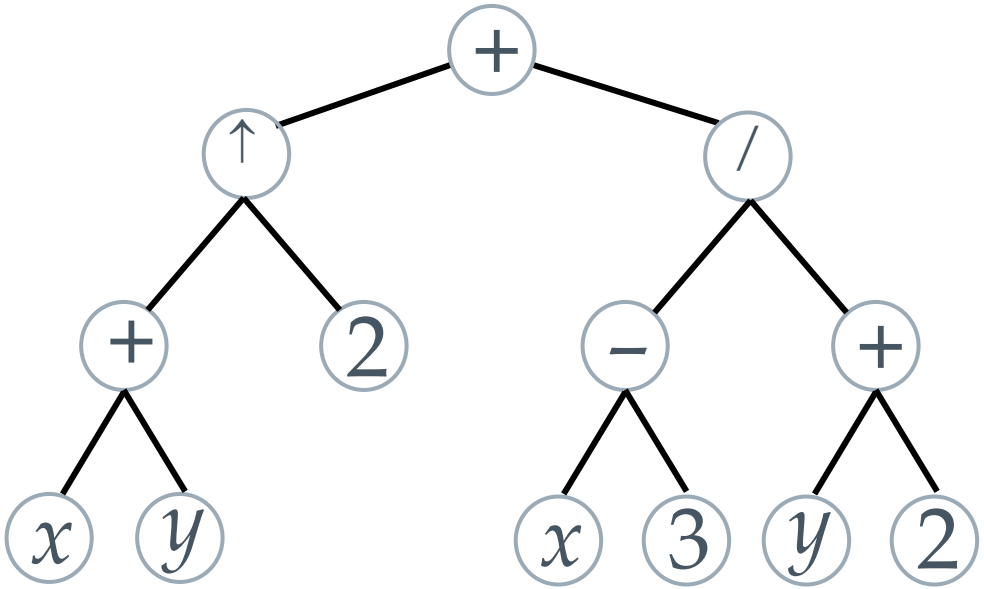
› Traverse in inorder (LVR) adding parentheses for each operation



$$(((x + y) \uparrow 2) + ((x - 3) / (y + 2)))$$

Prefix Notation (Polish Notation)

› Traverse in preorder (VLR)



+ ↑ + x y 2 / – x 3 + y 2

Evaluating Prefix Notation

- In an prefix expression, a binary operator precedes its two operands
- The expression is evaluated right-left
- Look for the first operator from the right
- Evaluate the operator with the two operands immediately to its right

Example

+ / + 2 2 2 / – 3 2 + 1 0

+ / + 2 2 2 / – 3 2 1

+ / + 2 2 2 / 1 1

+ / + 2 2 2 1

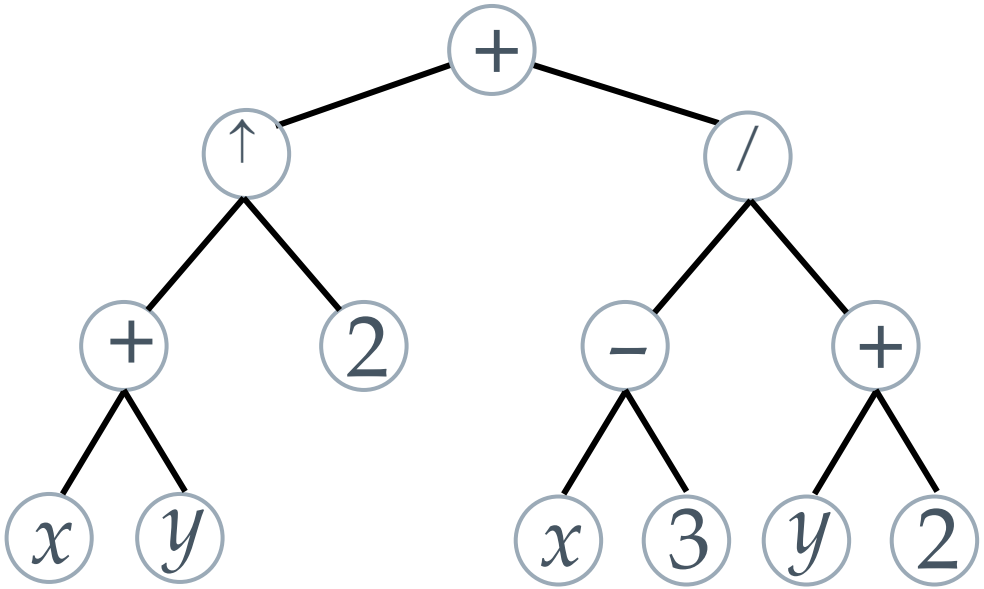
+ / 4 2 1

+ 2 1

3

Postfix Notation (Reverse Polish)

› Traverse in postorder (LRV)



$x\ y\ +\ 2\ \uparrow\ x\ 3\ -\ y\ 2\ +\ /\ +$

Evaluating Postfix Notation

- In an postfix expression, a binary operator follows its two operands
- The expression is evaluated left-right
- Look for the first operator from the left
- Evaluate the operator with the two operands immediately to its left

Example

2 2 + 2 / 3 2 – 1 0 + / +

4 2 / 3 2 – 1 0 + / +

2 3 2 – 1 0 + / +

2 1 1 0 + / +

2 1 1 / +

2 1 +

3

User-defined Data Type

› Use an aggregate structure to define new type, in C:

```
struct EmployeeType
{
  char    Name[25];
  int     Age;
  real    SkillRating;
}
```

› Use the new type to define variables:

```
struct EmployeeType DistManager, SalesRep1;
```

Abstract Data Type

- A user-defined data type that can include both data (representation) and functions (behavior).
- Example:

```
interface StackType
{
    public int pop();
    public int push(int item);
    public boolean isEmpty();
    public boolean isFull();
}
```

Class

- An abstract data type with extra features.
  - Properties can be inherited.
  - Constructor methods to initialize new objects.
  - Contents can be encapsulated.

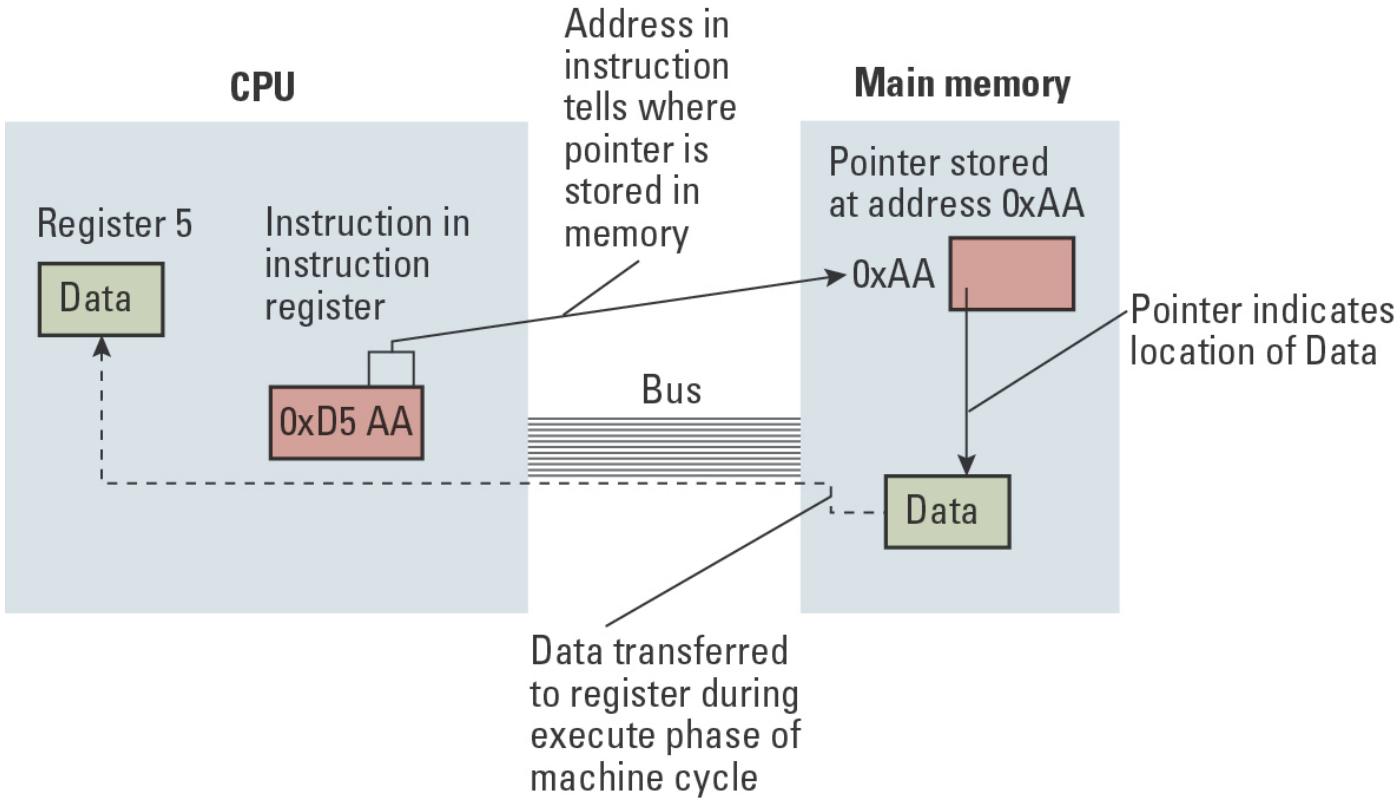
A Stack of Integers Implemented in Java and C#

```
class StackOfIntegers implements StackType

{

    private int[] StackEntries = new int[20];

    private int StackPointer = 0;
    public void push(int NewEntry)
    {   if (StackPointer < 20)

            StackEntries[StackPointer++] = NewEntry;

    }
    public int pop()
    {   if (StackPointer > 0) return StackEntries[--StackPointer];

        else return 0;

    }
    public boolean isEmpty()
    {   return (StackPointer == 0);

    }
    public boolean isFull()
    {   return (StackPointer >= MAX);

    }

}
```

Pointers in Machine Language

- **Immediate addressing:** Instruction contains the data to be accessed.
- **Direct addressing:** Instruction contains the address of the data to be accessed.
- **Indirect addressing:** Instruction contains the location of the address of the data to be accessed.

Our First Attempt at Expanding the Machine Language in Appendix C to Take Advantage of Pointers



Loading a Register from a Memory Cell that is Located by Means of a Pointer Stored in a Register