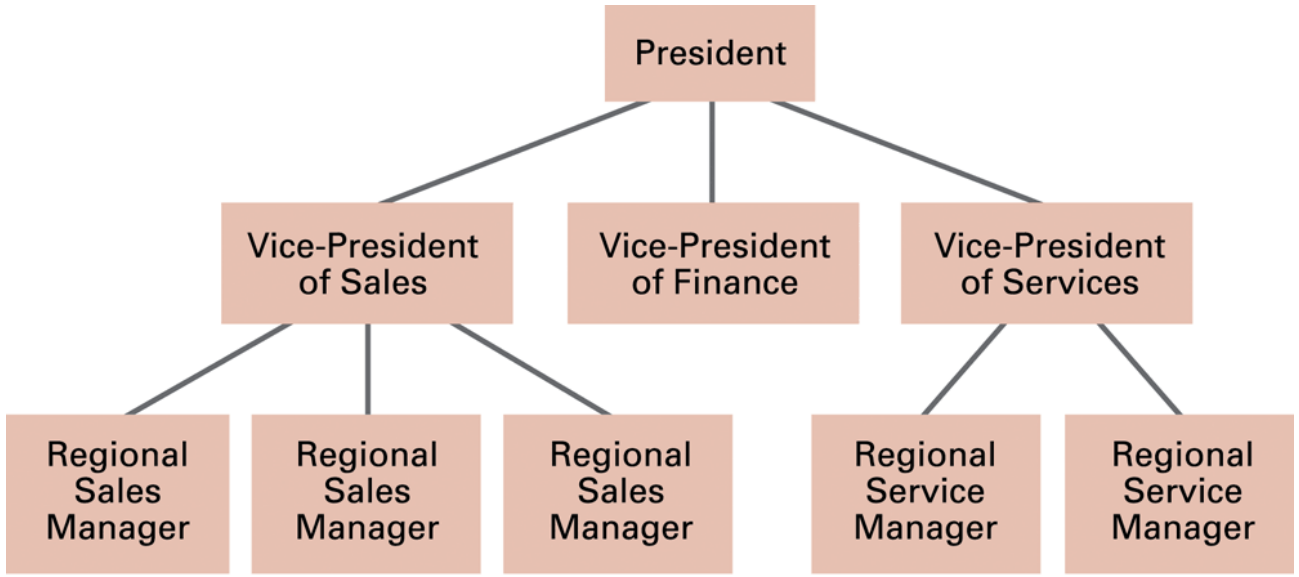
Data Abstractions

National Chiao Tung University
Chun-Jen Tsai
05/23/2012

Concept of Data Structures

- How do we store some conceptual structure in a linear memory?
- For example, an organization chart:



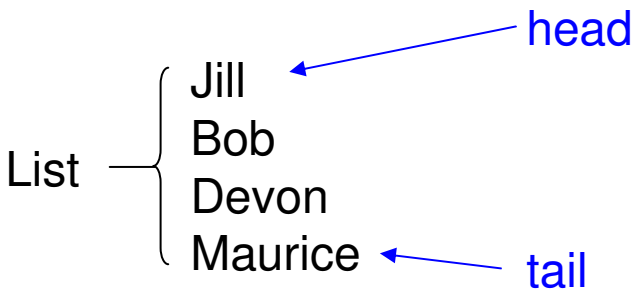
Basic Data Structures

- As discussed in Programming Language, **arrays** and **structures** are two aggregate data types directly supported by most programming languages
- Some other commonly used data types are
  - List
    - Stack (Last-In-First-Out, LIFO)
    - Queue (First-In-First-Out, FIFO)
  - Tree

⇨ We have to find an efficient way to *implement* these data types! (By implement we mean mapping of these data types to the memory cells of a computer)

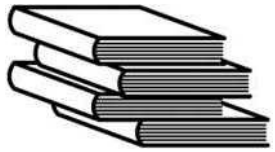
Terminology for Lists

- A **list** is a collection of data whose entries are arranged sequentially
- The beginning of a list is called the **head**; and the end of a list is called the **tail**

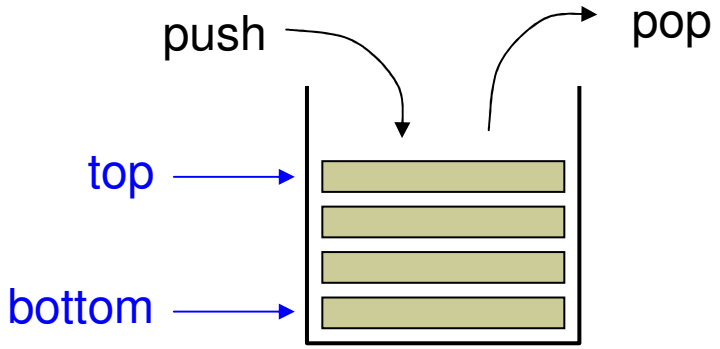


Terminology for Stacks

- A **stack** is a list in which entries are removed and inserted only at the head
- Data are entered into a stack in a last-in-first-out (**LIFO**) manner
- Insert an item at the top of a stack is **push**, removing an item from the top of a stack is **pop**

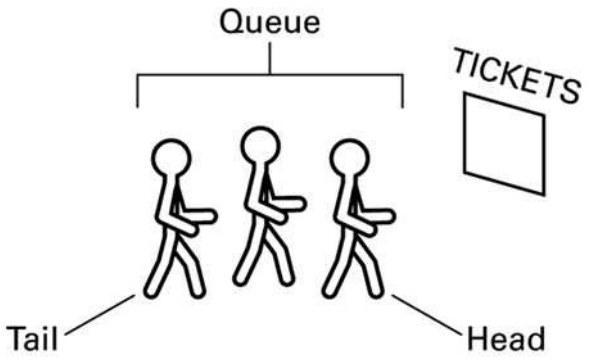


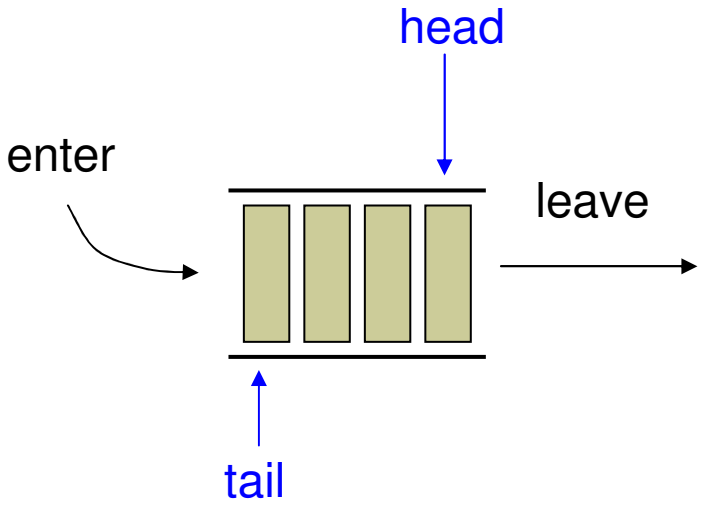
a stack of books



Terminology for Queues

- A **queue** is a list in which entries are removed at the head and are inserted at the tail
- Data are entered into a stack in a first-in-first-out (**FIFO**) manner





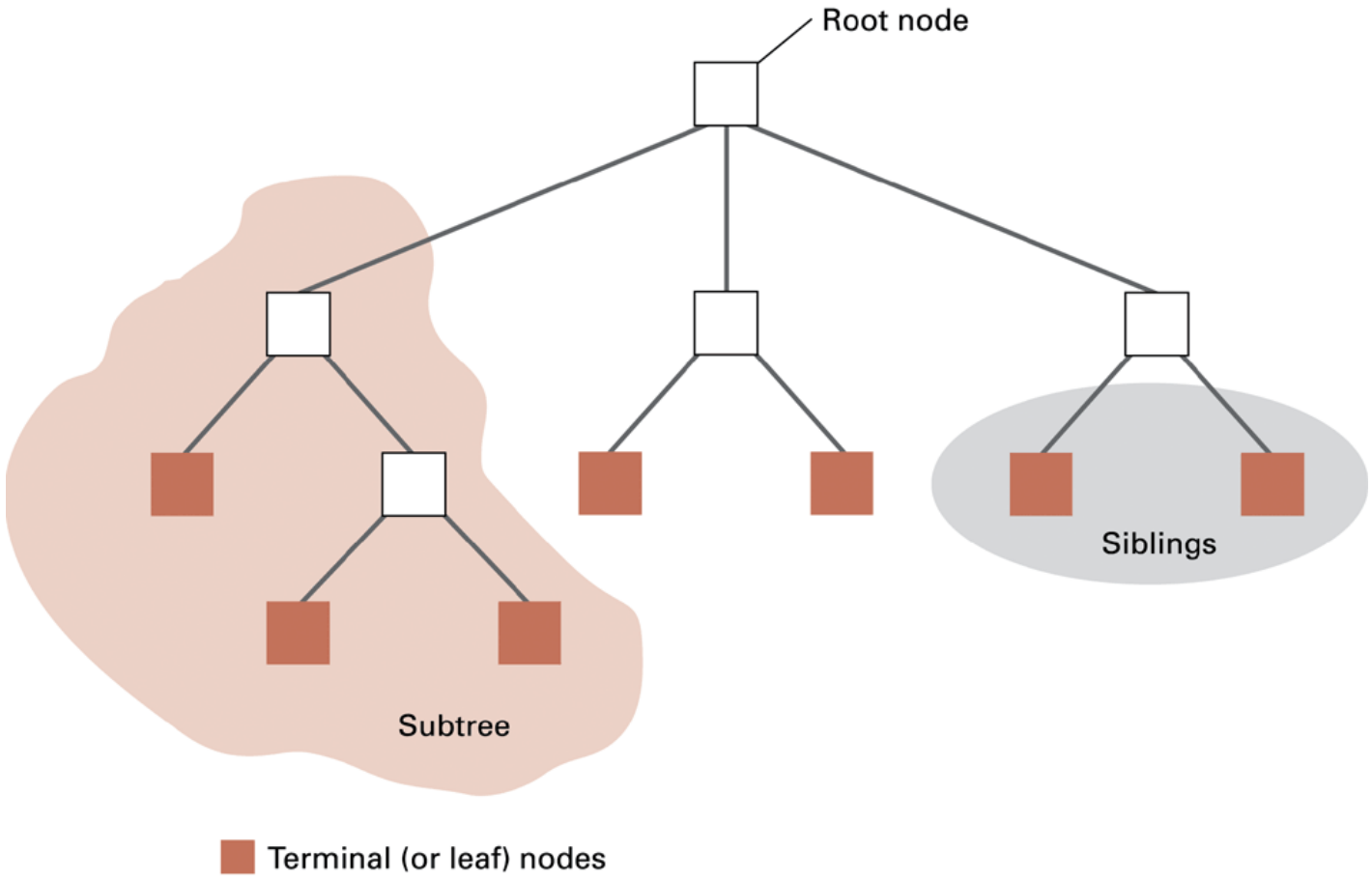
Terminology for Trees (1/2)

- A **tree** is a collection of data whose entries have a hierarchical organization
- Each entry in a tree is called a **node**
- The **root** is the node at the top of the tree
- A **leaf** (or **terminal**) is a node at the bottom of the tree
- The **parent** of a node is the node immediately above the specified node; a **child** of a node is a node immediately below the specified node

Terminology for Trees (2/2)

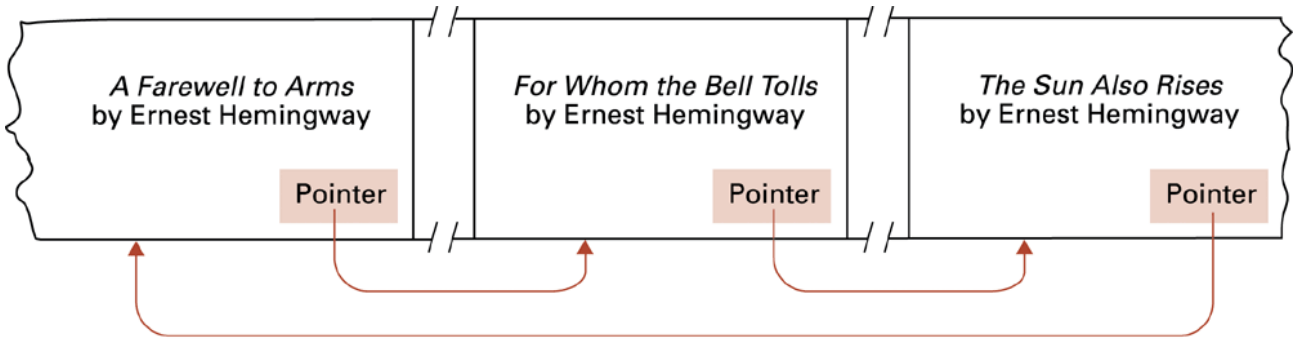
- The **ancestors** of a node is the node's parent, parent of parent, etc.
- The **descendent** of a node is the node's children, children of children, etc.
- The **siblings** of a node is all the nodes that share the same parent with that node
- A **binary tree** is a tree in which every node has at most two children
- The **depth** of a tree is the number of nodes in the longest path from the root to the leaf

Example: A Tree



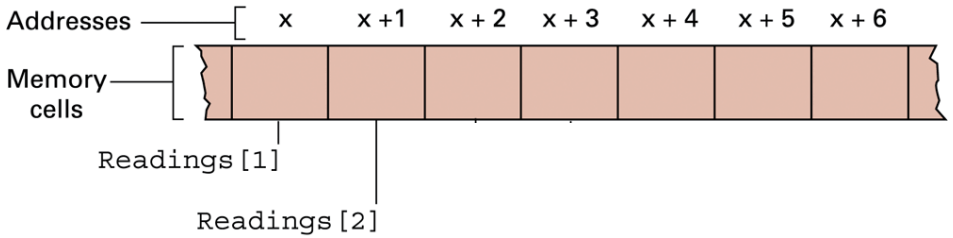
Characteristics of Data Structures

- Static data structure:
  - The size of the data structure does not change
- Dynamic data structure:
  - The size of the data structure can change
- Pointer:
  - The address of a data item in memory cell; a pointer is used to locate a data item in memory



Storing Arrays

- Homogeneous arrays are usually stored in contiguous memory blocks

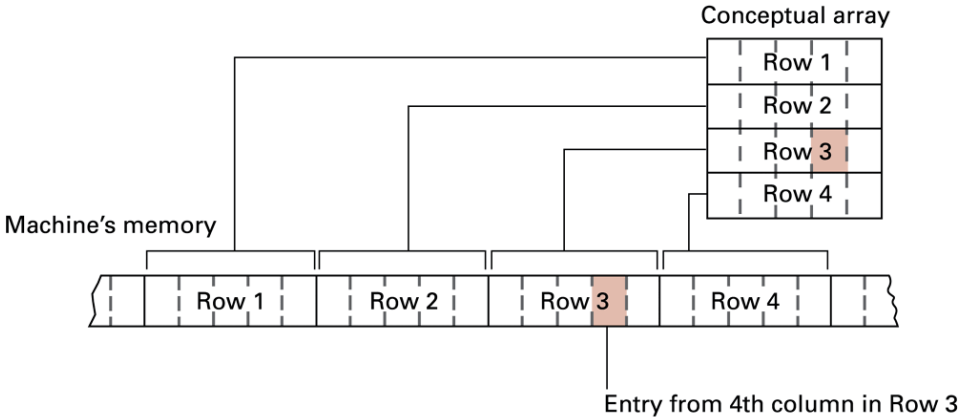


- Heterogeneous arrays (structures) can be stored in contiguous blocks or non-contiguous blocks using pointers

```
Struct {
    char Name[8];
    int Age;
    float SkillRating;
} Employee;
```

Storing Two Dimensional Arrays

- Memory cells are in one-dimensional order, to store a two-dimensional array, some mapping must be done:
  - **Row-major**: data entries are stored row-by-row
    - Address polynomial: $A[i, j] = \chi + c \times (i-1) + (j-1)$

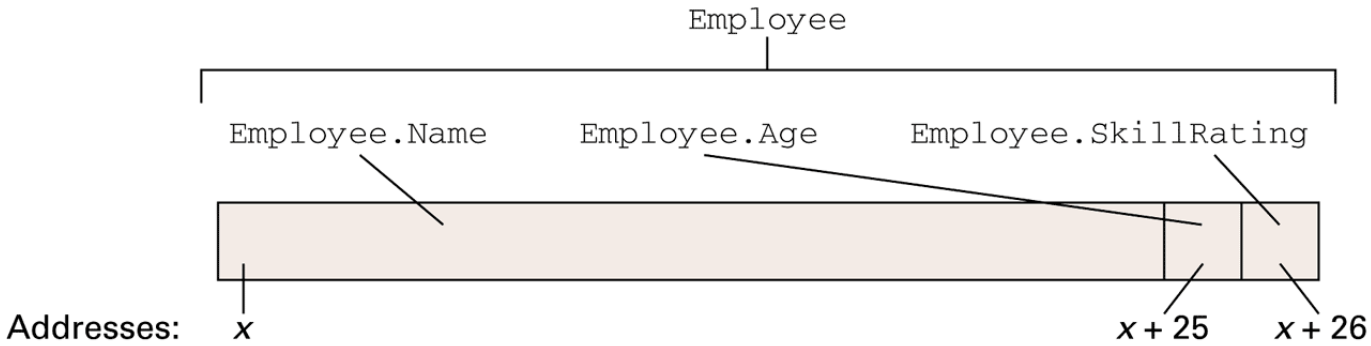


  - **Column-major**: data entries are stored column-by-column
    - Address polynomial: $A[i, j] = m + (i-1) + r \times (j-1)$

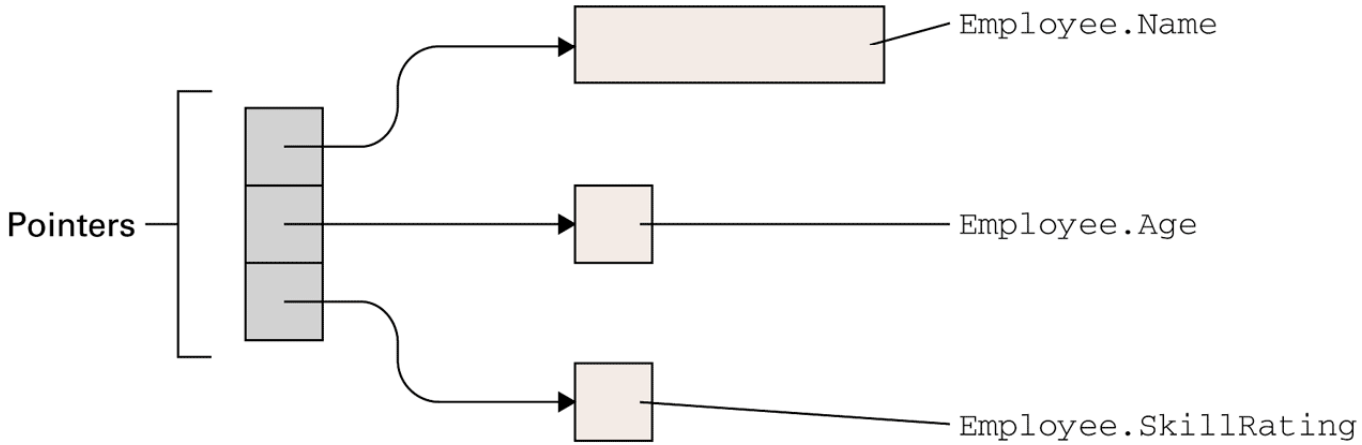
Note: m is the address of the first data item, i is the row index, j is the column index, c is the number of columns, and r is the number of rows

Storing Heterogeneous Arrays

- Two ways to store a heterogeneous array



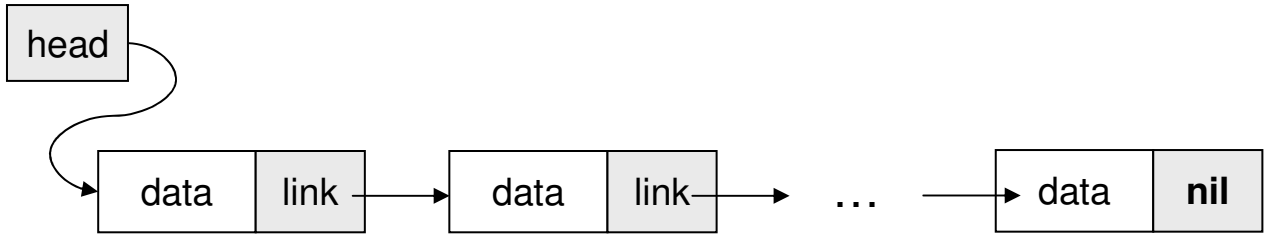
a. Array stored in a contiguous block



b. Array components stored in separate locations

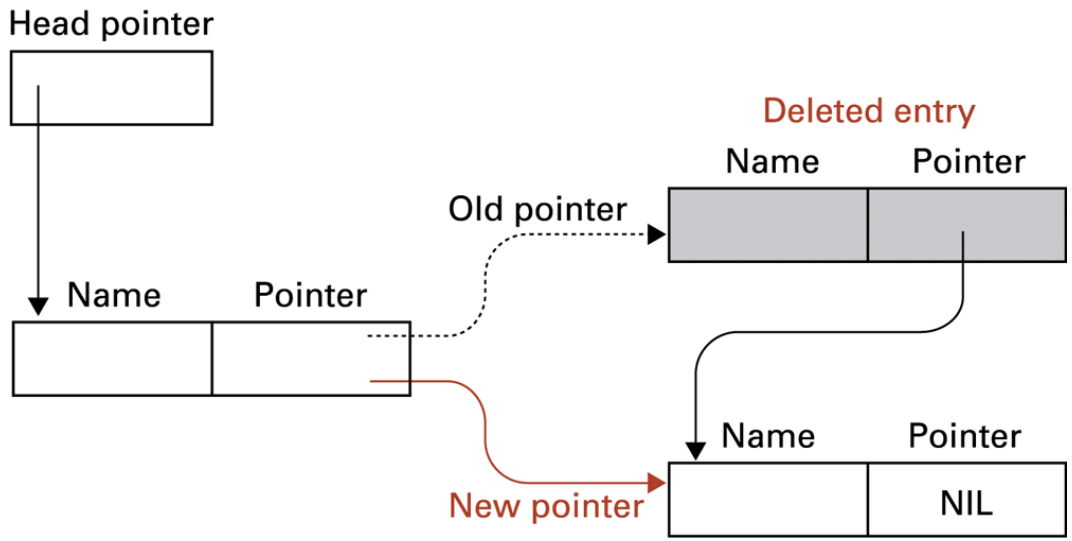
Storing Lists

- A list which is stored in a homogeneous array is called a contiguous list
- A list in which each node points to the next one is called a linked list
  - A **head pointer** is a pointer to the first entry of the list
  - A **nil pointer** is a special value used to indicate the end of the list



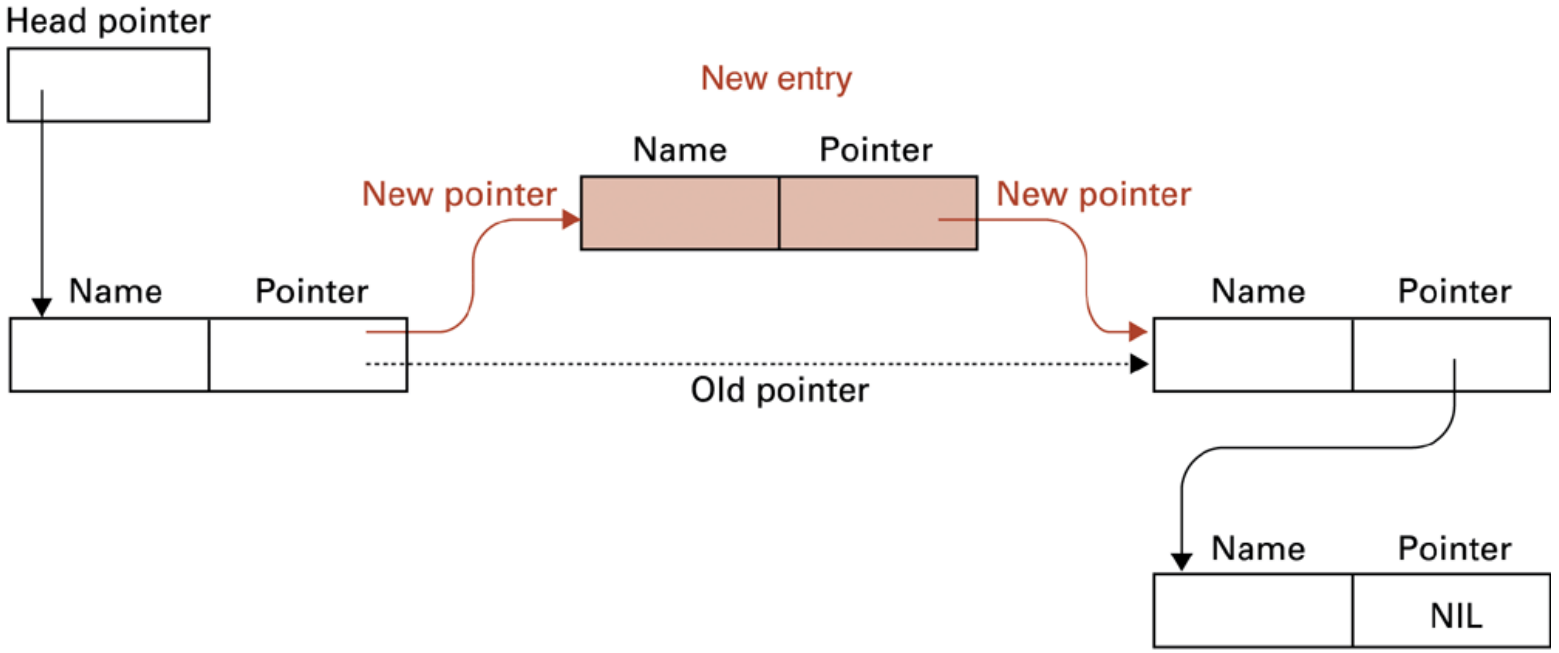
Manipulating Link Lists (1/2)

- Deleting an entry



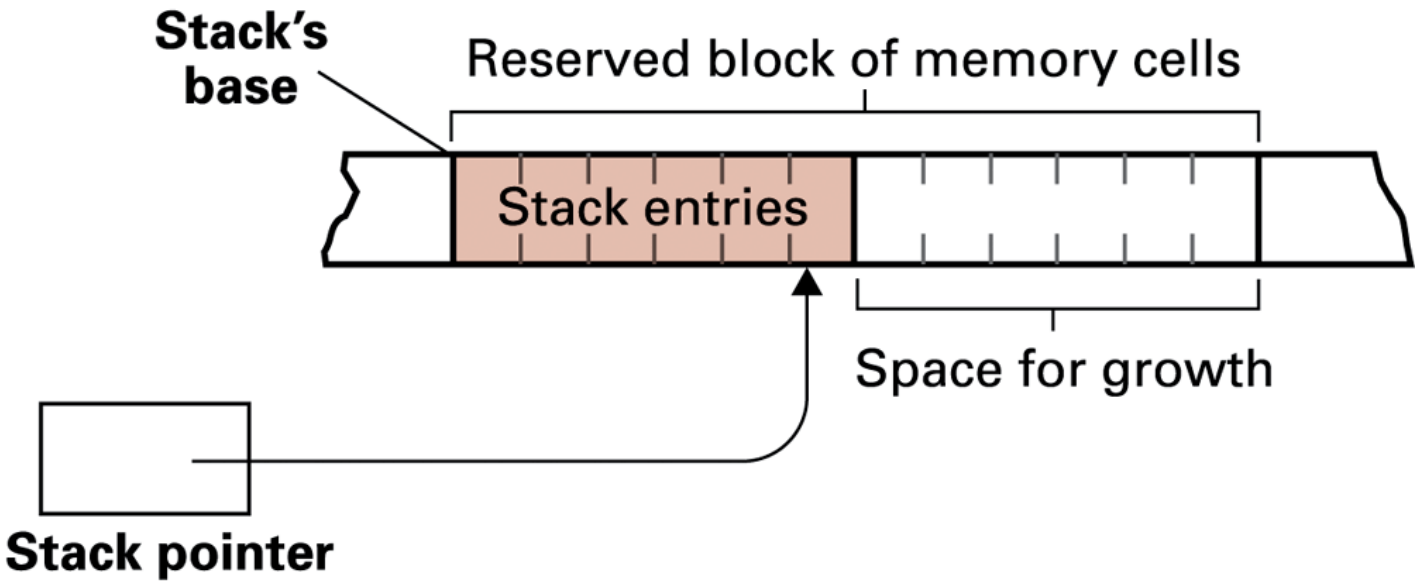
Manipulating Link Lists (2/2)

- Inserting an entry



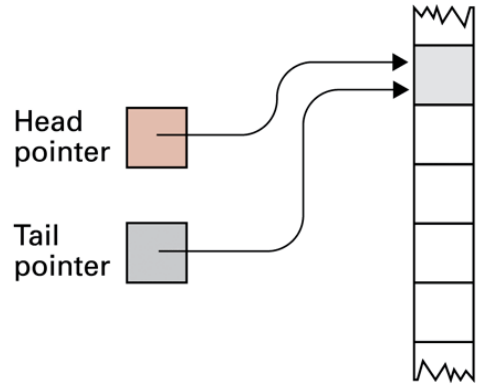
Storing Stacks and Queues (1/2)

- Both stacks and queues can be stored using same mechanisms as those for lists
- For example, for stacks contiguous lists can be used:

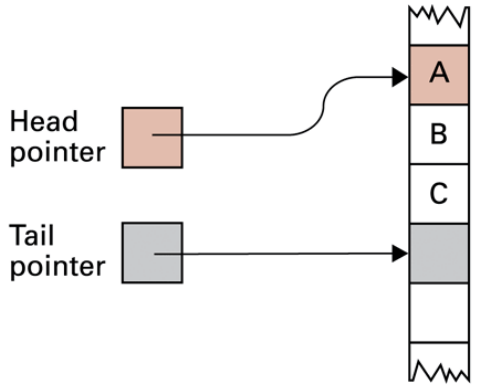


Storing Stacks and Queues (2/2)

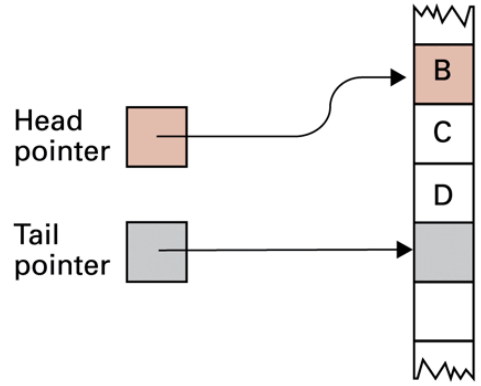
- We can also uses a contiguous list to store queues:



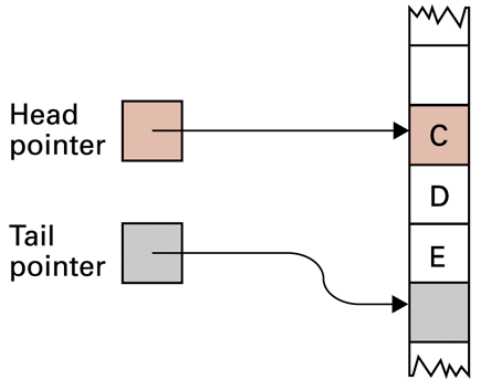
a. Empty queue



b. After inserting entries A, B, and C



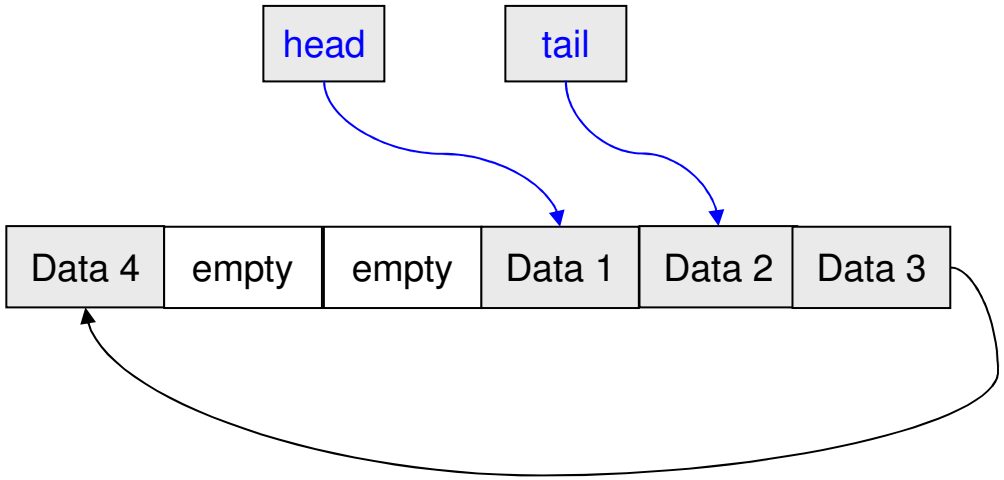
c. After removing A and inserting C and D



d. After removing B and inserting E

Circular Queues

- Queues are often stored as **circular queues** too:
  - Stored in an array where the first entry is considered to follow the last entry
  - Prevents a queue from crawling out of its allotted storage space



Storing a Binary Tree

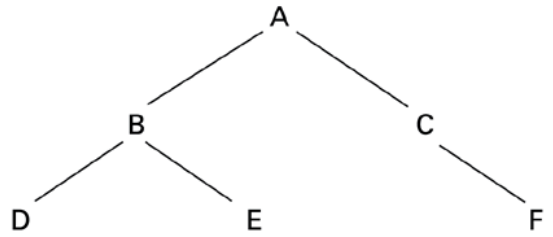
- Again, we can use linked structure or contiguous array to store a binary tree:
- Linked structure
  - Each node contains a data cell and two child pointers
  - Accessed through a pointer to root node

- Contiguous array
  - A[1] = root node
  - A[2],A[3] = children of A[1]
  - A[4],A[5],A[6],A[7] = children of A[2] and A[3]
  - …

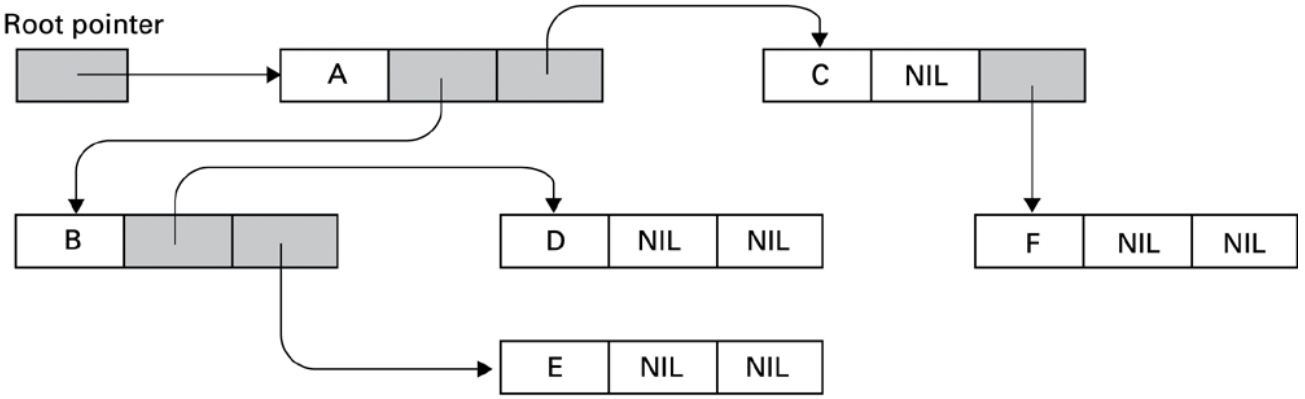
Linked Binary Tree

- The conceptual and actual organization of a binary tree using a linked storage system are as follows:

Conceptual tree



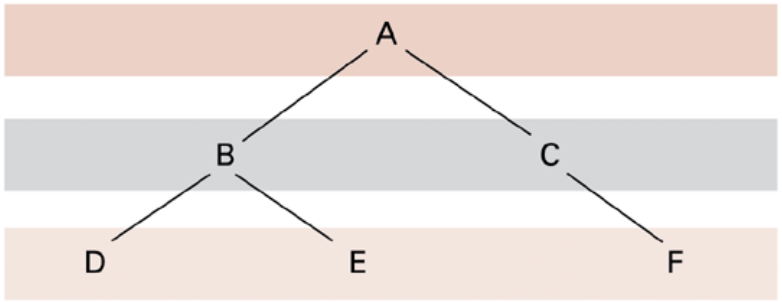
Actual storage organization



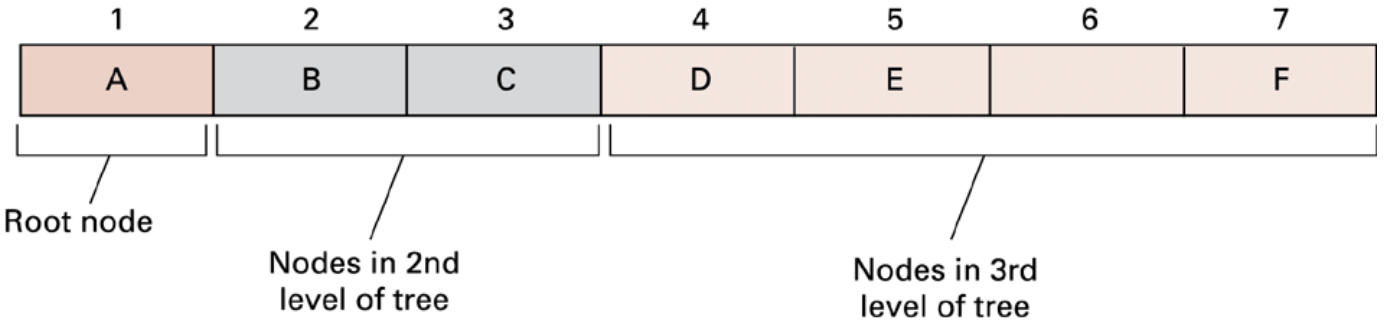
Binary Tree Array

- A binary tree stored in a contiguous array:

Conceptual tree

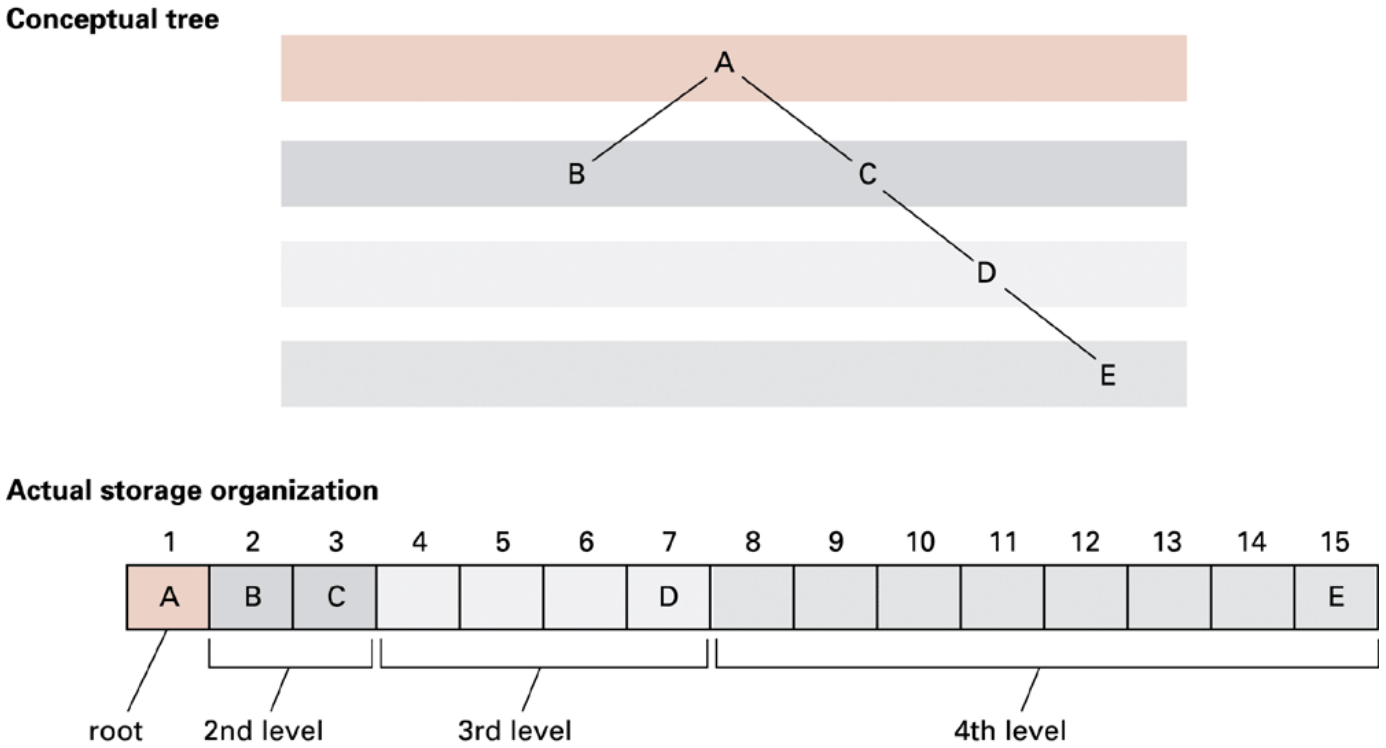


Actual storage organization



Drawbacks of Using Arrays for Trees

- A sparse, unbalanced tree shown in its conceptual form and as it would be stored without pointers:



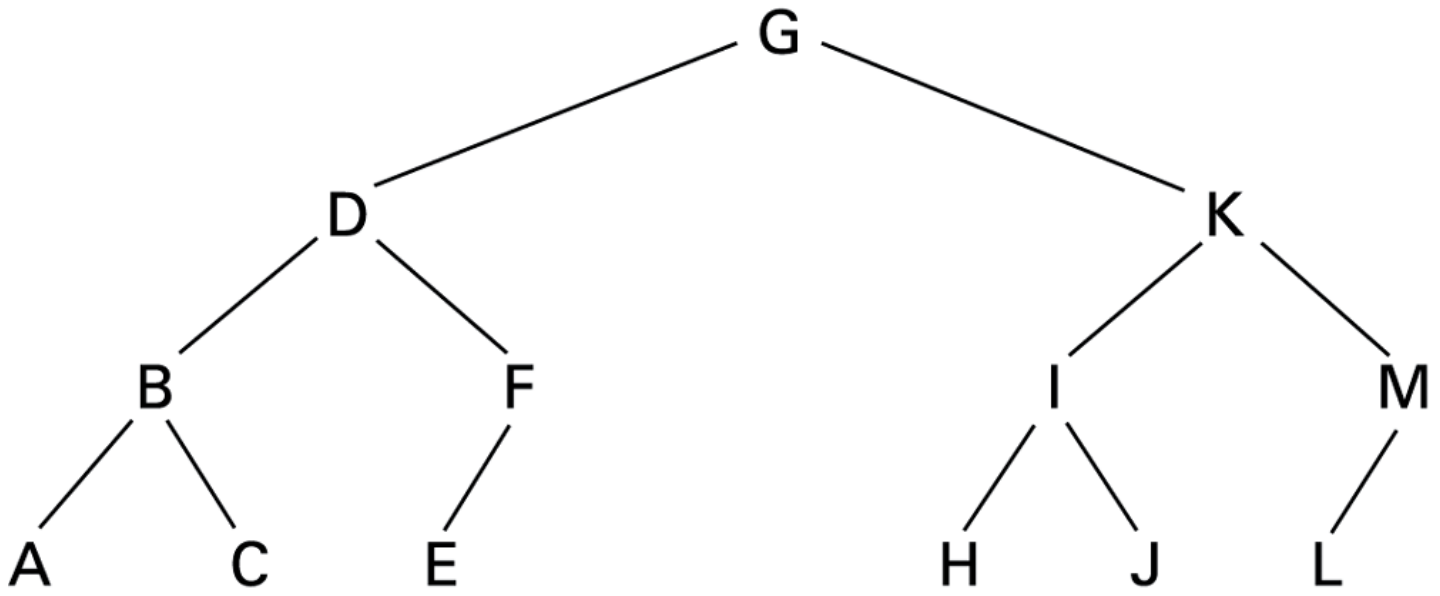
Manipulating Data Structures

- Ideally, a data structure should be manipulated solely by pre-defined procedures.
  - Example: A stack typically needs at least `push` and `pop` procedures
  - The data structure along with these procedures constitutes a complete abstract tool
- Example: print a linked list

```
procedure PrintList (List)
CurrentPointer ← head pointer of List.
while (CurrentPointer is not NIL) do
    (Print the name in the entry pointed to by CurrentPointer;
     Observe the value in the pointer cell of the List entry
      pointed to by CurrentPointer, and reassign CurrentPointer
      to be that value.)
```

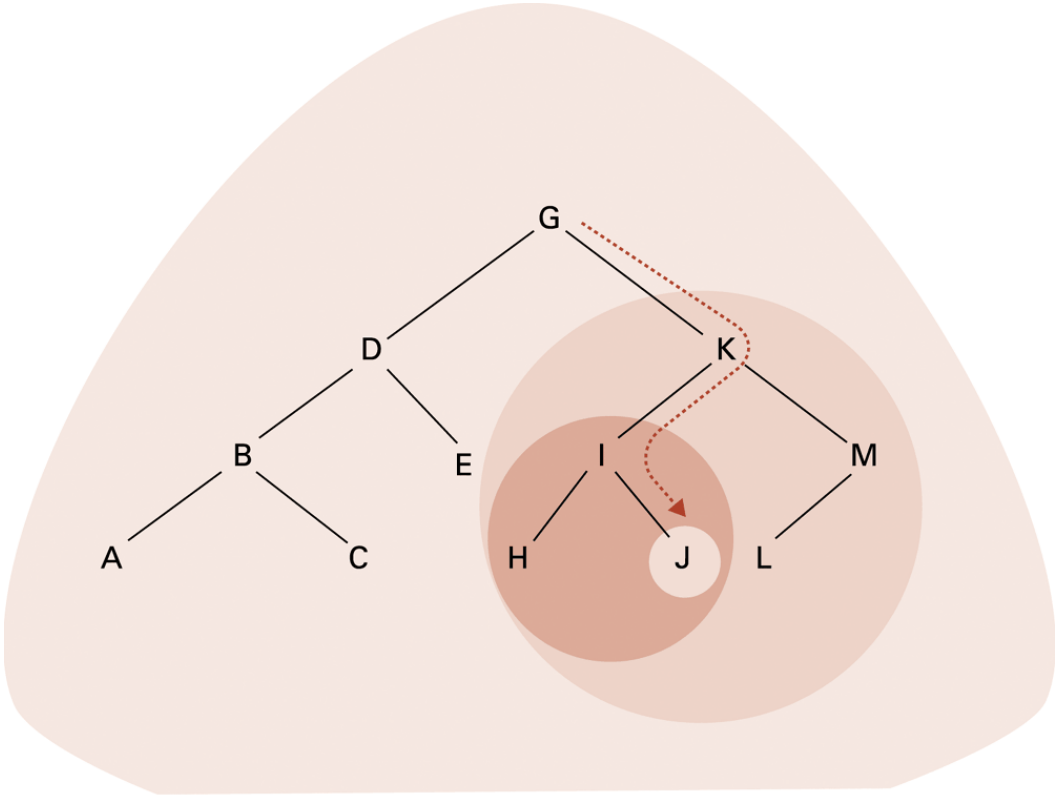
Ordered Tree

- An ordered is a tree where the noses are sorted (e.g. from left-to-right):



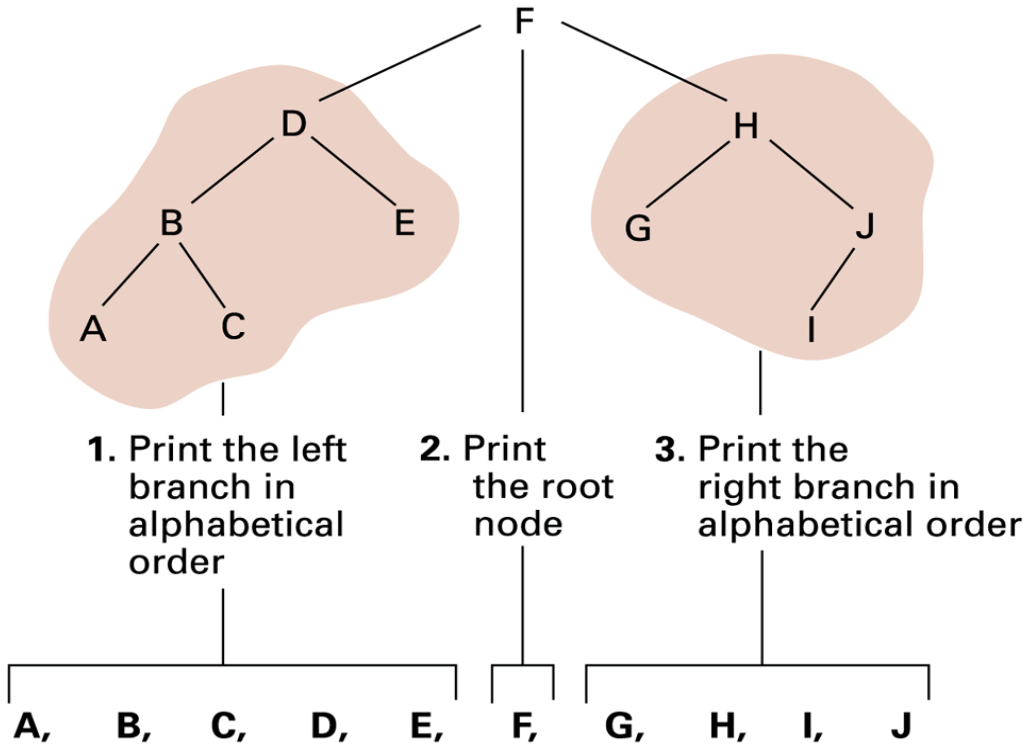
Tree Traversal (1/2)

- Traversing a binary ordered tree is just like performing a binary search



Tree Traversal (2/2)

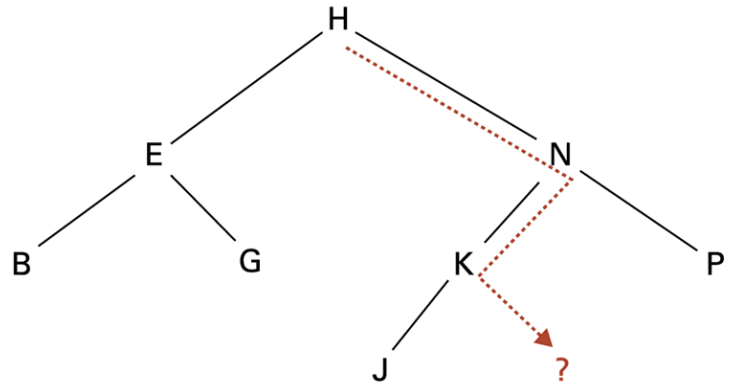
- We can print a search tree in alphabetical order using tree traversal as well:



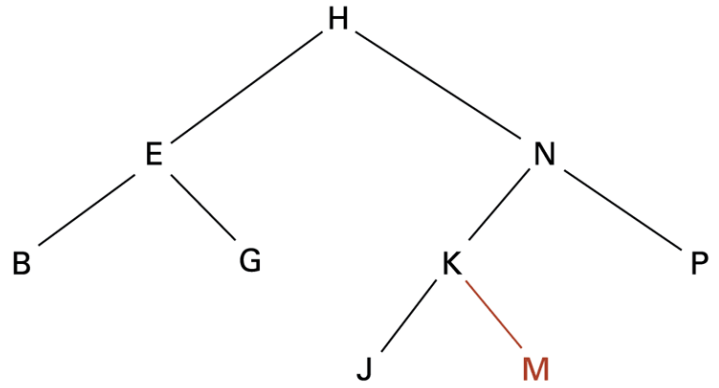
Inserting a Node

- Inserting the entry ‘M’ into the ordered binary 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



User-Defined Data Types

- Objected-oriented paradigm allows the definition of a new data type and its valid operations. For example, a stack of integers in Java and C# is as follows:

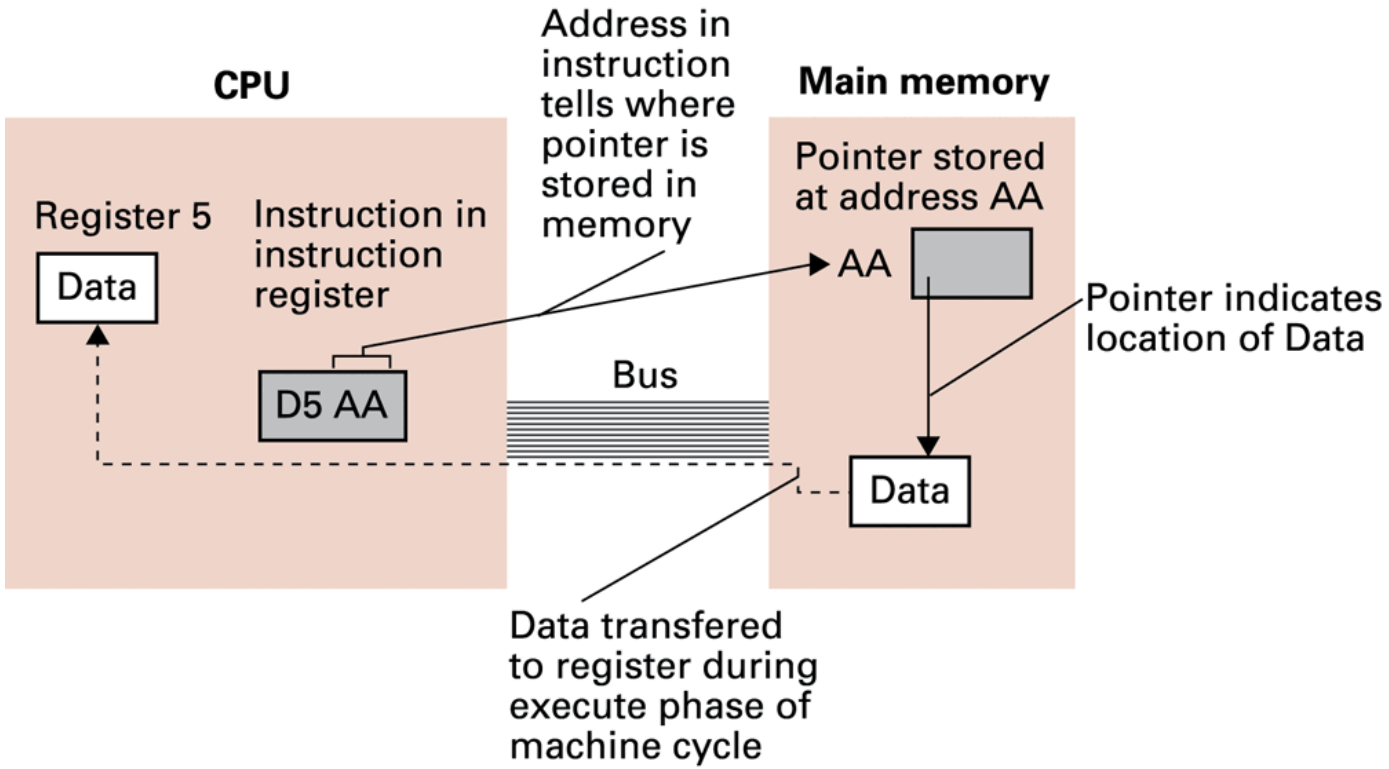
```
class StackOfIntegers
{private int[] StackEntries = new int[20];
 private int StackPointer = 0;


 public void push(int NewEntry)
 {if (StackPointer < 20)
    StackEntries[StackPointer++] = NewEntry;
 }

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

Data Accessing in Machine Code

- 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 address (location) of memory cells*** that has the address of the data to be accessed
  - Such a memory cell is often called a “pointer”

Ex: Machine in Appendix C (1/2)

- Loading data from memory via indirect addressing; with a memory cell as the pointer



Ex: Machine in Appendix C (2/2)

- Loading data from memory via indirect addressing; with a register as the pointer