Intelligent Data & Coding Structures

Dr. Cahit Karakuş
Istanbul, Turkey

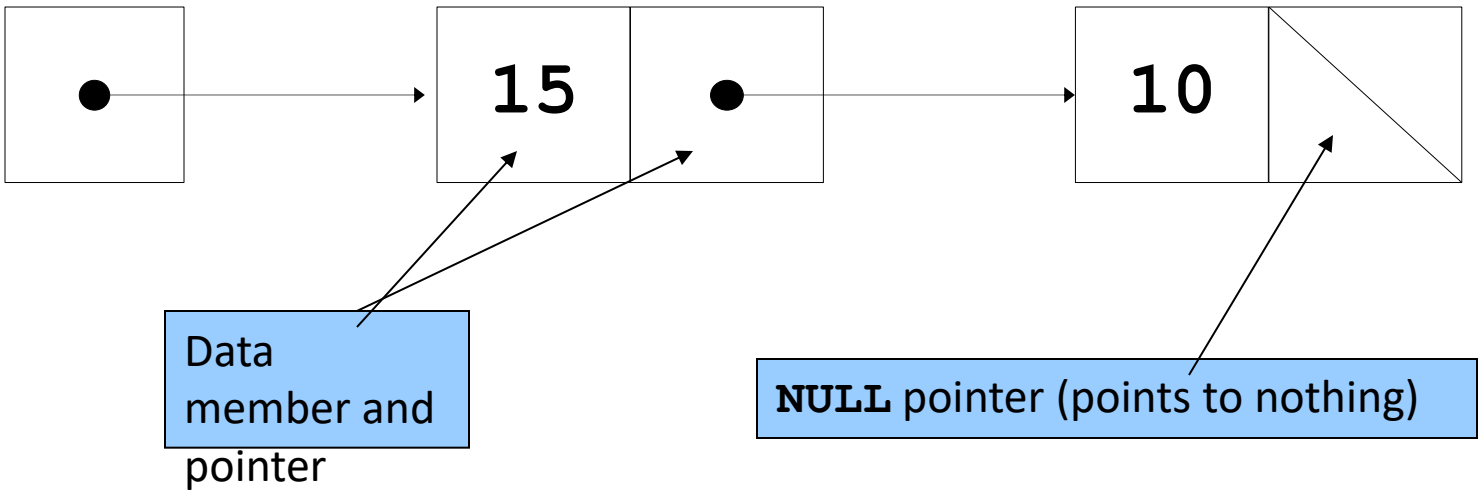
Data Structures

Introduction

- Dynamic data structures
  - Data structures that grow and shrink during execution
- Linked lists
  - Allow insertions and removals anywhere
- Stacks
  - Allow insertions and removals only at top of stack
- Queues
  - Allow insertions at the back and removals from the front
- Binary trees
  - High-speed searching and sorting of data and efficient elimination of duplicate data items

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **`NULL`** pointer (**0**)
- Diagram of two self-referential structure objects linked together



Self-Referential Classes

```
struct node {
    int data;
    struct node *nextPtr;
}
```

- **`nextPtr`**
  - Points to an object of type node
  - Referred to as a link
    - Ties one **`node`** to another **`node`**

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- **`malloc`**
  - Takes number of bytes to allocate
    - Use **`sizeof`** to determine the size of an object
  - Returns pointer of type **`void *`**
    - A **`void *`** pointer may be assigned to any pointer
    - If no memory available, returns **`NULL`**
  - Example

```
newPtr = malloc( sizeof( struct node ) );
```

- **`free`**
  - Deallocates memory allocated by **`malloc`**
  - Takes a pointer as an argument
  - **`free ( newPtr );`**

Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to null to mark the list's end
- Use a linked list instead of an array when
  - You have an unpredictable number of data elements
  - Your list needs to be sorted quickly

Linked Lists

- Types of linked lists:
  - Singly linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - Circular, singly linked
    - Pointer in the last node points back to the first node
  - Doubly linked list
    - Two “start pointers” – first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - Circular, doubly linked list
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to **`NULL`**
  - Constrained version of a linked list
- **`push`**
  - Adds a new node to the top of the stack
- **`pop`**
  - Removes a node from the top
  - Stores the popped value
  - Returns **`true`** if **`pop`** was successful

Queues

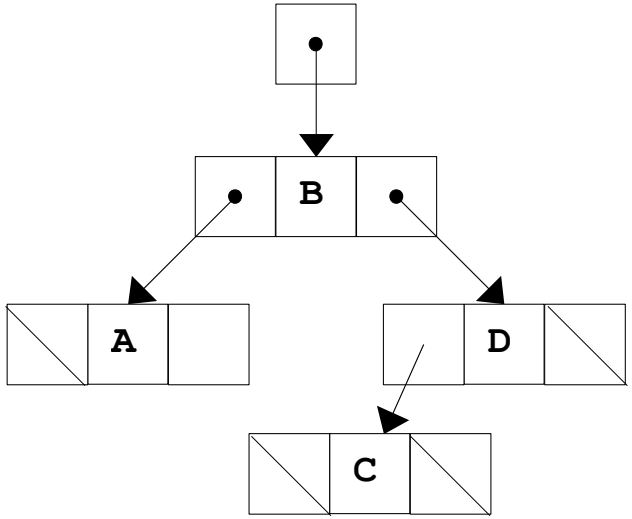
- Queue
  - Similar to a supermarket checkout line
  - First-in, first-out (FIFO)
  - Nodes are removed only from the head
  - Nodes are inserted only at the tail
- Insert and remove operations
  - Enqueue (insert) and dequeue (remove)

Trees

- Tree nodes contain two or more links
  - All other data structures we have discussed only contain one
- Binary trees
  - All nodes contain two links
    - None, one, or both of which may be NULL
  - The root node is the first node in a tree.
  - Each link in the root node refers to a child
  - A node with no children is called a leaf node

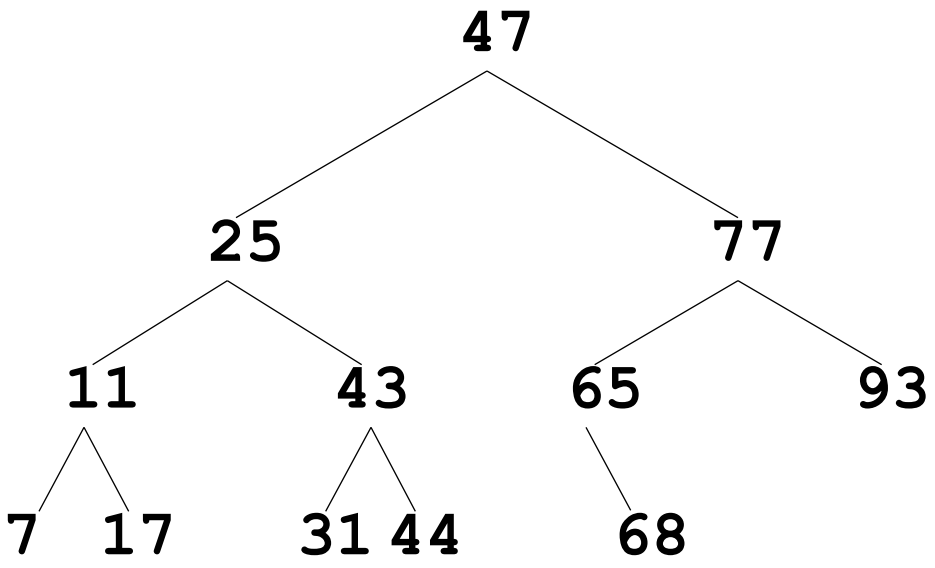
Trees

- Diagram of a binary tree



Trees

- Binary search tree
  - Values in left subtree less than parent
  - Values in right subtree greater than parent
  - Facilitates duplicate elimination
  - Fast searches - for a balanced tree, maximum of $\log_2 n$ comparisons



Trees

- Tree traversals:
  - Inorder traversal – prints the node values in ascending order
    1. Traverse the left subtree with an inorder traversal
    2. Process the value in the node (i.e., print the node value)
    3. Traverse the right subtree with an inorder traversal
  - Preorder traversal
    1. Process the value in the node
    2. Traverse the left subtree with a preorder traversal
    3. Traverse the right subtree with a preorder traversal
  - Postorder traversal
    1. Traverse the left subtree with a postorder traversal
    2. Traverse the right subtree with a postorder traversal
    3. Process the value in the node