Cape Peninsula University of Technology

Software Design 2
SDN260S

Custom Generic Data Structures

H. Mataifa

Department of Electronic, Electrical and Computer Engineering

Outline

- Data structures: background
- Self-referential class
- Linked list
- Stack
- Queue
- Tree

Background

- **Data structures in programming**:

  - In the realm of computer science, data structures are specialized formats for organizing, storing, and managing data. They determine how data is stored in memory, allowing for efficient access and manipulation. The choice of an appropriate data structure can significantly impact the efficiency and performance of a program.

- **Types of data structures include:**

1. **Arrays** are collections of elements, each identified by an index. They provide fast access to elements but are less efficient when it comes to inserting or deleting elements.

2. **Linked Lists** are composed of nodes, each containing data and a reference to the next node. They are efficient for inserting and deleting elements but offer slower access times compared to arrays.

3. **Stacks** are a last-in, first-out (**LIFO**) data structure, often used for managing function calls, undo operations, and parsing expressions.

4. **Queues** are a first-in, first-out (**FIFO**) data structure, suitable for tasks like scheduling, task management, and data buffering.

5. **Trees** are hierarchical data structures with a root node and child nodes, allowing for efficient searching and sorting. Examples include binary trees and **AVL** trees.

6. **Graphs** consist of nodes and edges and are used to represent complex relationships and connections in various applications, such as social networks and network routing.

7. **Hash Tables** provide fast data retrieval based on a key. They are used extensively in databases and dictionaries.

Background

- **Importance of data structures**:

  - **Efficiency**: data structures are fundamental to achieving efficient data processing. They enable algorithms to run faster and consume fewer resources by optimizing data storage and access patterns.

  - **Problem Solving**: many computer science problems require the manipulation of data. Data structures provide a systematic way to approach these problems and devise efficient solutions.

  - **Code Reusability**: well-designed data structures can be reused across different projects, saving time and effort in software development.

  - **Memory Management**: data structures help manage memory efficiently, reducing memory leaks and optimizing resource allocation.

  - **Algorithm Design**: algorithms often rely on specific data structures. Choosing the right data structure can make algorithm development and optimization more straightforward.

  - **Scalability**: as data sizes grow, the choice of data structure becomes critical. A poorly chosen data structure can lead to performance bottlenecks in large-scale applications

Background

- **Key characteristics of data structures**:
  - **Linear vs. nonlinear**: data elements can be arranged sequentially/linearly or non-sequentially
  - **Static vs. dynamic**: size and memory location of static data structures is fixed at compile time. Dynamic data structures grow and shrink as required by the program, memory location is not fixed
  - **Homogeneous vs. non-homogeneous**: data elements in homogeneous data structures are of the same type (e.g. array), need not be of the same type in non-homogeneous data structures (e.g. struct)
  - **Time complexity**: time required to execute certain operations on the data structure (e.g. insertion/deletion are fast on a linked list, slower on an array)
  - **Space complexity**: memory requirements for certain operations on a data structure (e.g. nonlinear data structures such as trees are generally more memory-efficient than linear data structures such as lists)

Background

- **Main applications of data structures**:
  - **Storing data** (e.g. database management system, use of hash table)
  - **Managing resources and services** (e.g. process scheduling queue, file directory management tree)
  - **Ordering, sorting, searching** (binary search tree for sorting and searching, priority queue for priority-based data management)
  - **Data exchange** (information such as TCP/IP packets is organized using data structures)
  - **Indexing** (B-trees for indexing data items stored in a database)

- **Most frequent operations on data structures**:
  - **Searching** (attempt to locate a specific data item within a data structure)
  - **Sorting** (arranging data elements in a data structure in a certain order)
  - **Insertion** (adding a data item to a data structure)
  - **Deletion** (removing a data item from a data structure)
  - **Updating** or replacing a part of a data structure

Self-Referential Class

- A self-referential class contains an instance variable that refers to another object of the same class type (e.g. generic **Node** class below)

```
class Node< T >
{
   private T data;
   private Node< T > nextNode; // reference to next linked node

   public Node( T data ) { /* constructor body */ }
   public void setData( T data ) { /* method body */ }
   public T getData() { /* method body */ }
   public void setNext( Node< T > next ) { /* method body */ }
   public Node< T > getNext() { /* method body */ }
} // end class Node< T >
```

Fig. 1: Self-referential Node class

- **Node** class has two instance variables (one of which is of the same type as the class being defined):
  - Generic Object **T** to hold the Node data
  - **Node** Object to hold the link to the next Node
- Self-referential class is used to constitute a link node in dynamic data structures such as Linked Lists

Linked List

- A linked list is a linear collection (i.e. a sequence) of self-referential class objects, called **nodes**, connected by reference links.
- Linked list has first node object used to access it in a program; each subsequent node is accessed via reference link from previous node; link in last node is set to null
- Linked list vs. array:
  - Array is static, linked list is dynamic (length of list can increase/decrease as needed)
  - Data access in array is generally faster than in linked list, but element insertion/removal is faster in linked list

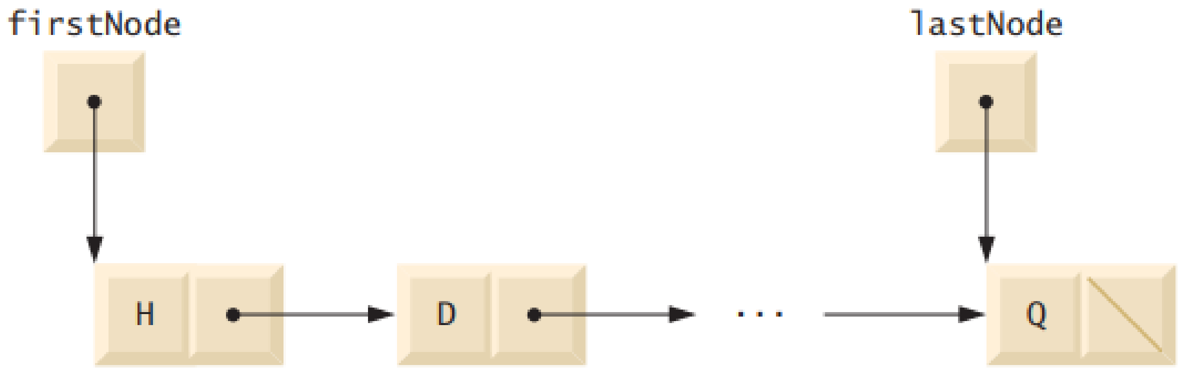

Fig. 2: Linked list graphical representation

Generic List Class

- Generic class **List<T>** (Fig. 22.3) comprises two (generic) **Node<T>** objects to represent first node and last node of the list
- **List<T>** class has methods to insert a node, remove a node, and to print the data stored in the nodes
- Node insertion and removal can be done at the back or front of the list
- **ListNode<T>** class comprises:
  - Generic instance variable to store node data
  - Node Object reference to store link to next node
  - Two constructors
  - Methods to get node data and next node

Stack

- A stack is a constrained version of a list. Whereas a new node can be inserted anywhere and an existing node removed from anywhere in a list, insertion and removal of nodes takes places from only one side of the stack – the top of the stack

- A stack is thus referred to as a Last-In, First-Out (**LIFO**) data structure – the last item to be added to a stack will also be the first one to be removed from it

- Primary methods for manipulating a stack are push and pop. Method push adds a new node to the top of a stack, method pop removes a node from the top of a stack (and returns the data stored in the node)

- Stacks are extensively used in system applications, e.g. program-execution stack used for keeping track of how functions interact with one another, and the state of each actively executing function
  - When a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the program-execution stack
  - Program-execution stack also contains the memory for local variables on each invocation of a method. When the method returns to its caller, the memory for the local variables is popped off the stack

- **Section 22.5** (Textbook) implements a custom stack data structure, first by inheritance (from generic class **List<T>**) then by composition (of **List<T>** object within class **Stack**)

- Advantage of composition over inheritance (for this specific example) is the ability to “hide” class **List<T>**’s methods that should not be accessible in class **Stack**

Queue

- A queue represents a waiting line – the first client to join the line is serviced first. Node insertion is at the back (or tail) end of the queue, node removal is at the front (or head) end of the queue.
- A queue is thus referred to as a First-In, First-Out (**FIFO**) data structure – the first item to be added to a queue will also be the first one to be removed from it
- Primary methods for manipulating a queue are enqueue and dequeue. Method enqueue adds a new node to the back-end of a queue, method dequeue removes a node from the front-end of a queue
- Queue data structure has many applications in computer systems, for example:
  - In process scheduling, each application waits in line to receive **CPU** time
  - Print jobs can be sent simultaneously to a printer, but only one print job can be executed at a time, so print jobs have to be queued as they wait for the printer to become available
  - Information packets in a computer network wait in a queue to be routed via network nodes to the right destination
  - File-access requests are enqueued as they wait for the file server to become available to handle the request
- **Section 22.6** (Textbook) implements a custom queue data structure, by composition (of **List<T>** object within class **Queue**)
- Using composition to adopt features of **List<T>** in class **Queue** enables hiding **List<T>**'s methods that should not be accessible in class **Queue**

Tree

- A tree is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links.
- There are many types of trees, the most prevalent being a binary tree, whose nodes contain two links (one or both of which may be null)
- Root node is the first node in a tree. Each link in the root node refers to a child. Left child is first node in left subtree, right child is first node in right subtree.
- Children of a specific node are referred to as siblings. A node with no children is called a leaf node
- **Binary search tree**: has the characteristic that values in any left subtree are less than those in the subtree's parent node, and values in any right subtree are greater than those in the subtree's parent node

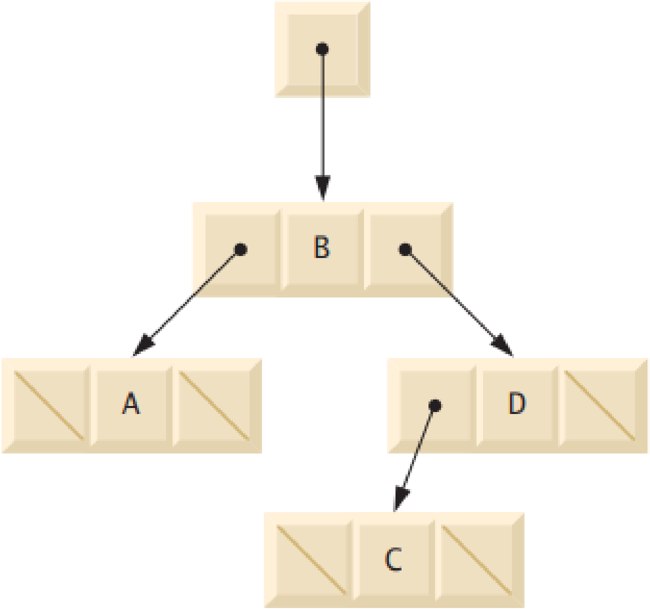

Fig. 3: Binary tree graphical representation

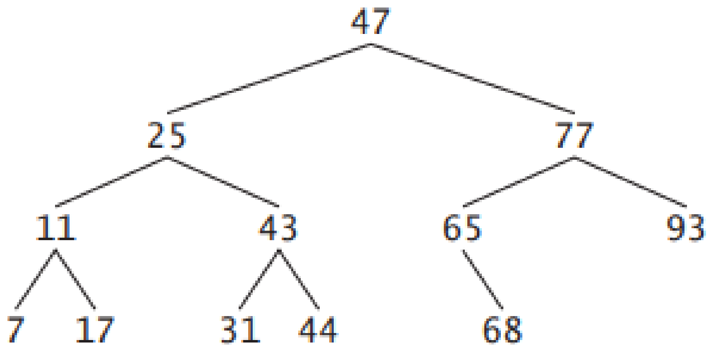

Fig. 4: Example of binary search tree

Tree

- **Section 22.7** (Textbook) implements a custom binary search tree data structure.
- The generic binary search tree includes three different tree traversal methods:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal
- Generic class **Tree<T>** makes use of self-referential class **TreeNode<T>** to implement the node object of the tree, which must extend interface **Comparable** so that data items can be compared to those existing in the tree before being added to the tree