DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

Lecture 4

Lists, Stacks and Queues

Outlines of Class (Lecture 4)

- Introduction
- List
- List Implementation
  - ✓ Array-based List
  - ✓ Linked List
- Doubly Linked Lists
- Stack
  - ✓ Array Based Stack
  - ✓ Linked Stack
- Queue
- Dictionaries

Introduction

- An abstract data type (ADT) is a set of objects together with a set of operations.
- Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, reals, and booleans are data types.
- The C++ class allows for the implementation of ADTs, with appropriate hiding of implementation details.
- The following three data structures that we will study in this lecture are primary examples of ADTs.
  - ✓ List
  - ✓ Stack
  - ✓ Queue

Lists

- Position (a first element in the list, a second element, and so on.)
- Finite, ordered sequence of data items known as elements (not sorted by value)
- The list ADT: used for lists of integers, lists of characters, lists of payroll records
- Empty (List contains no elements)
- Length of the list (The number of elements currently stored)
- Head (The beginning of the list)
- Tail (The end of the list)
- Relationship between the value of an element and its position in the list
  - ✓ sorted lists (elements positioned in ascending order of value)
  - ✓ unsorted lists (no particular relationship between element values and positions)

Lists (cont.)

- To indicate the list of four elements, with the current position being to the right of the bar at element 12
  - <20, 23 | 12, 15>
- Calling insert with value 10
  - <20, 23 | 10, 12, 15>
- getValue method
  - Returns a pointer to the current element

ADT for a LIST

```
template <typename E> class List {
private:
   void operator =(const List&) {  }
   List(const List&) {  }
public:
   List ( ) { }
   virtual ~List( ) { }
   virtual void clear() = 0;
   virtual void insert(const E& item) = 0;
   virtual void append(const E& item) = 0;
   virtual E remove( ) = 0;
   virtual void moveToStart( ) = 0;
   virtual void moveToEnd( ) = 0;
   virtual void prev( ) = 0;
   virtual void next( ) = 0;
   virtual int length( ) const = 0;
   virtual int currPos( ) const = 0;
   virtual void moveToPos(int pos) = 0;
   virtual const E& getValue( ) const = 0;
};
```

Iteration for a List

```
for (L.moveToStart(); L.currPos()<L.length(); L.next())
  {
        it = L.getValue();
        doSomething(it);
   }
```

Find Method

```
// Return true if "K" is in list "L", false otherwise

bool find(List<int>& L, int K)
{
int it;
for (L.moveToStart(); L.currPos()<L.length(); L.next())
{
it = L.getValue();
if (K == it) return true;              // Found K
}
return false;                          // K not found
}
```

Types of List

- Two standard approaches to implement lists
  - ✓ the array-based list
  - ✓ the linked list

Array-based List Implementation

- In Array-based List Implementation, AList
- Inherits from abstract class List
- Class AList's portion contains the data members for the array-based list.
- Allocated at some fixed size (the size of the array must be known when the list object is created)
- "=defaultsize" (the parameter is optional)
- Data members
  - ✓ listArray: the array which holds the list elements
  - ✓ maxSize: maximum permitted size
  - ✓ listSize: actual hold value (number of list items)
  - ✓ curr: current position

Array-based List Implementation (cont.)

- Class AList
- Stores the list elements in the first listSize contiguous array positions
- The head of the list is always at position 0.
- Access to any element using moveToPos method followed by getValue method takes $\Theta(1)$ time.
- To store list elements in contiguous cells of the array
  - ✓ Insert, append, and remove methods
- Append operation takes $\Theta(1)$ time.
- If an element is inserted at the head of the list, all elements currently in the list must shift one position toward the tail to make room
  - ✓ (n) time (n elements already in the list)
- To insert the position i within a list of n elements (n-i elements must shift toward the tail)
- To remove an element from the head of the list, all remaining elements must shift toward the head by one position
- To remove the element at position i, n- i -1 elements must shift toward the head.

Array-based List Implementation (cont.)

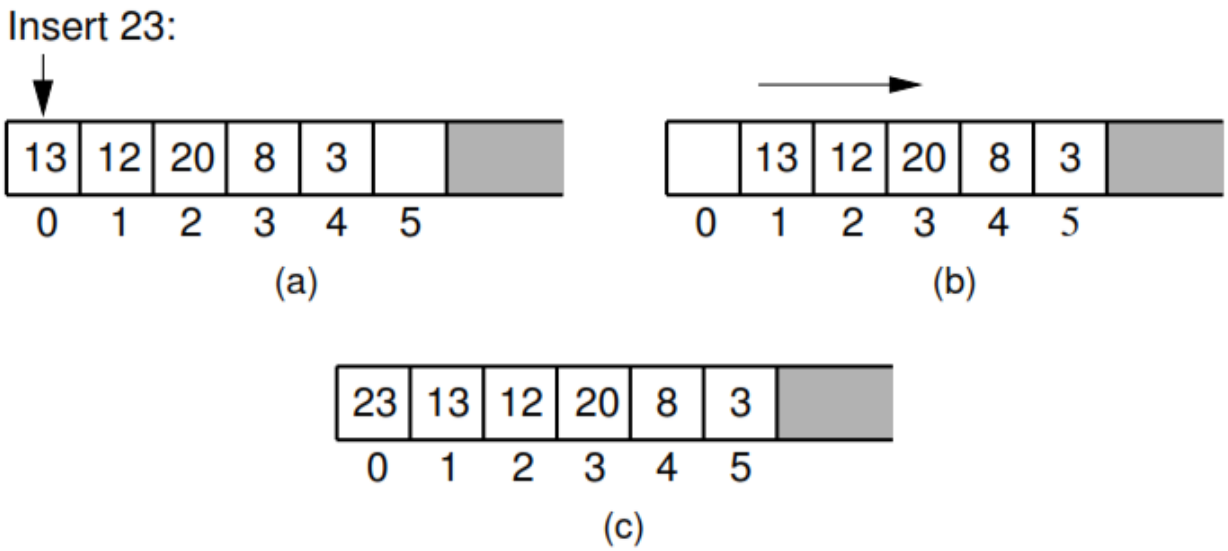


Figure 1: Inserting an element at the head of an array-based list requires shifting all existing elements in the array by one position toward the tail.
(a) A list containing five elements before inserting an element with value 23.
(b) The list after shifting all existing elements one position to the right.
(c) The list after 23 has been inserted in array position 0. Shading indicates the unused part of the array.

Linked Lists

- Use of pointers
- Use dynamic memory allocation (allocates memory for new list elements as needed)
- Made up of a series of objects called nodes of the list
- Linked List Node Implementation
  - ✓ Link class
  - ✓ Element field (to store the element value)
  - ✓ Next field (to store a pointer to the next node on the list)
  - ✓ NULL (points nowhere, i.e; for the last list node's next field)
  - ✓ head (point to the list's first node)
  - ✓ tail (point to the last link of the list)
  - ✓ curr (point to the current element)
  - ✓ Singly linked list or one-way list (each list node has a single pointer to the next node on the list)

Linked lists (cont.)

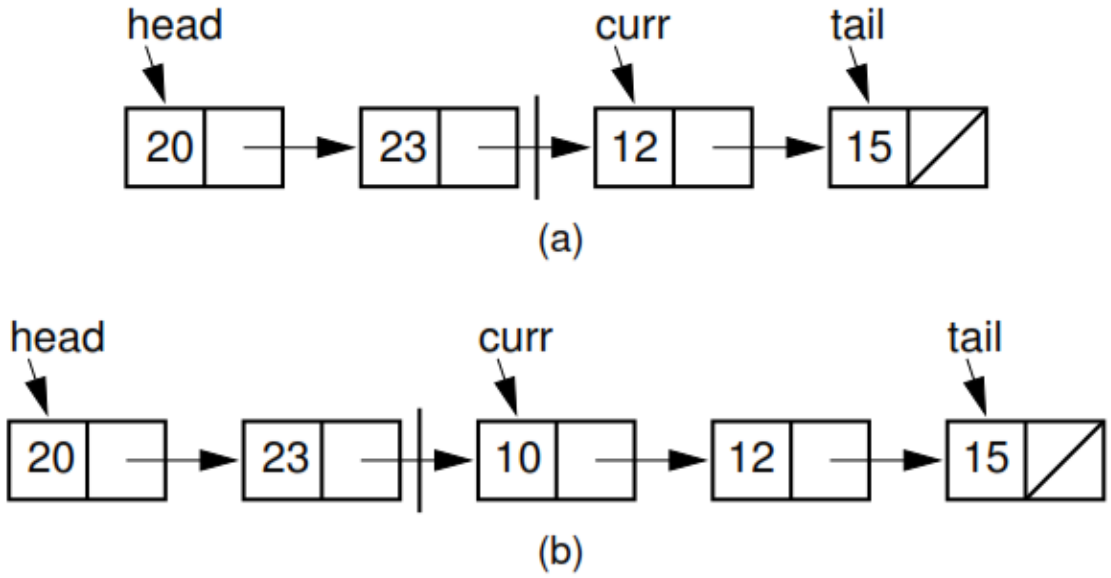


Figure 2: Illustration of a faulty linked-list implementation where curr points directly to the current node.
(a) Linked list prior to inserting element with value 10.
(b) Desired effect of inserting element with value 10.

Linked lists (cont.)

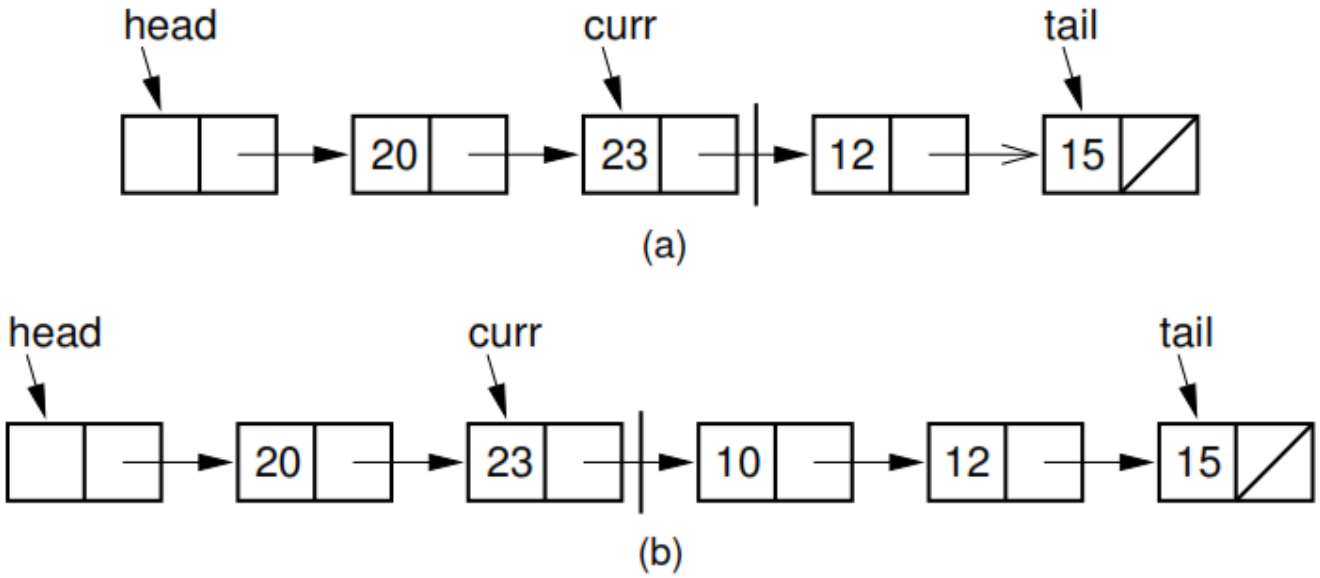


Figure 3: Insertion using a header node, with curr pointing one node head of the current element.
(a) Linked list before insertion. The current node contains 12.
(b) Linked list after inserting the node containing 10.

Linked lists (cont.)

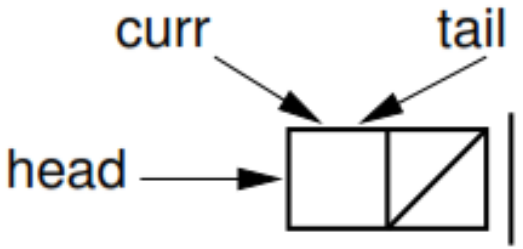


Figure 4: Initial state of a linked list when using a header node.

Linked List Insertion Process

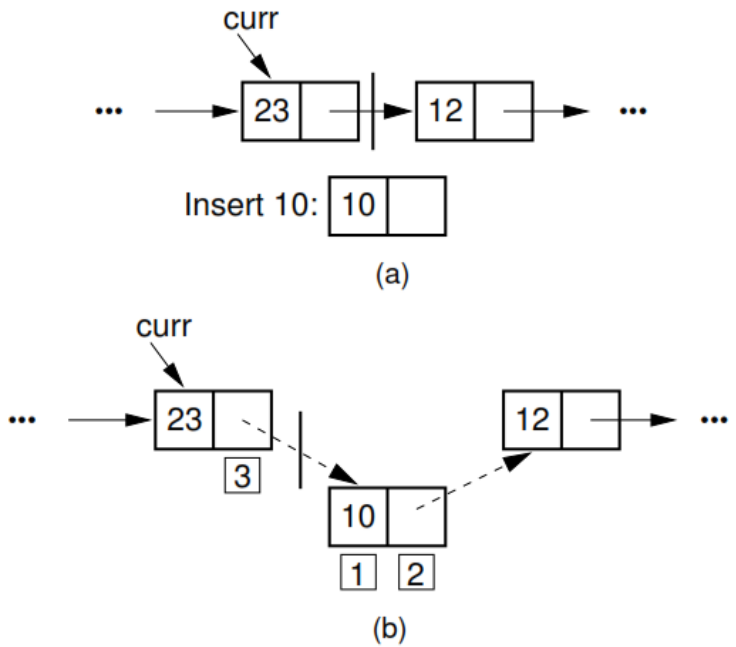


Figure 5: The linked list insertion process. (a) The linked list before insertion. (b) The linked list after insertion.

1. marks the element field of the new link node.
2. marks the next field of the new link node, which is set to point to what used to be the current node (the node with value 12).
3. marks the next field of the node preceding the current position. It used to point to the node containing 12; now it points to the new node containing 10.

Linked List Removal Process

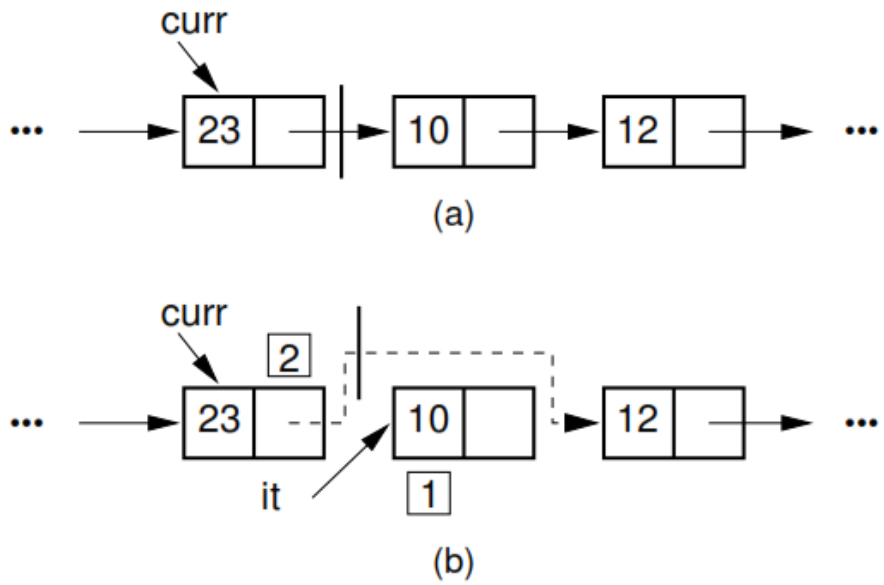


Figure 6: The linked list removal process. The linked list before removing the node with value 10. (b) The linked list after removal.

1. marks the list node being removed. **it** is set to point to the element.

2. marks the next field of the preceding list node, which is set to point to the node following the one being deleted.

Freelists

- Instead of making repeated calls to **new** and **delete**, the **Link** class can handle its own freelist.
- A freelist holds those list nodes that are not currently being used.
- When a node is deleted from a linked list, it is placed at the head of the freelist.
- When a new element is to be added to a linked list, the freelist is checked to see if a list node is available. If so, the node is taken from the freelist.
- If the freelist is empty, the standard **new** operator must be called.

Freelists are particularly useful for linked lists that periodically grow and then shrink.

Doubly Linked Lists

- The singly linked list allows for direct access from a list node only to the next node in the list.
- A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list.
- Two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.
- The header and the tailer: nodes that contain no value

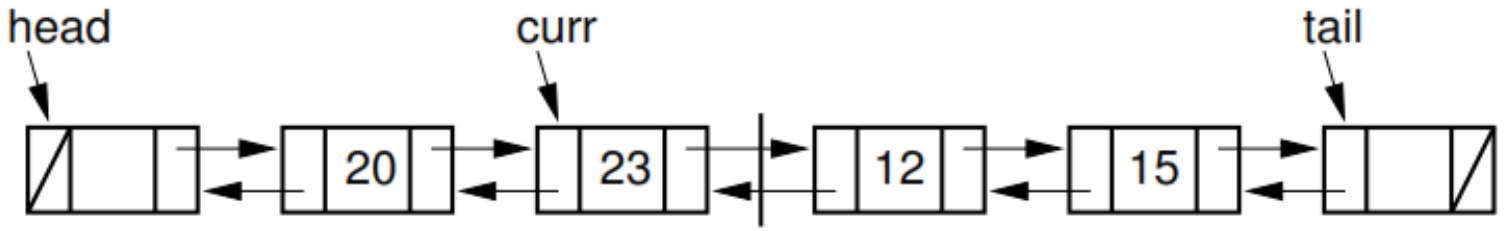


Figure 7: A doubly linked list

Insertion for Doubly Linked Lists

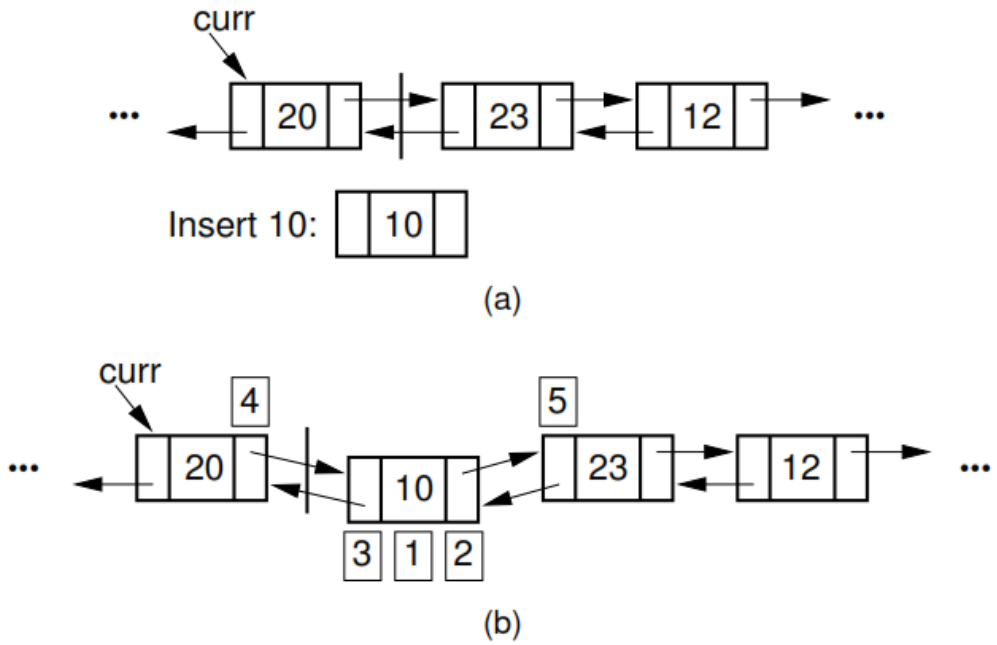


Figure 8: Insertion for doubly linked lists. The labels 1 , 2 , and 3 correspond to assignments done by the linked list node constructor. 4 marks the assignment to curr->next. 5 marks the assignment to the prev pointer of the node following the newly inserted node.

Doubly Linked List Removal

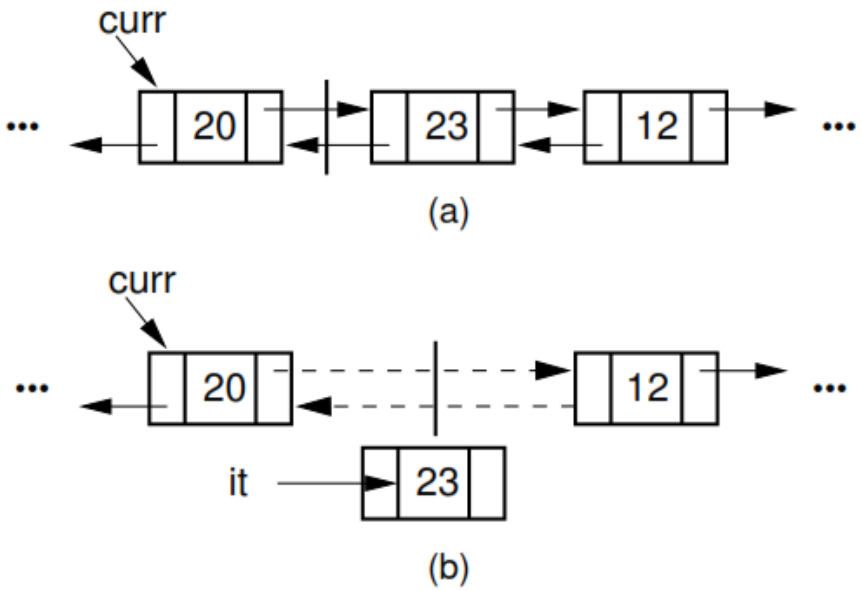


Figure 9: Doubly linked list removal. Element **it** stores the element of the node being removed. Then the nodes to either side have their pointers adjusted.

Stack

- A stack is a sequence of items (list-like structure) that are accessible at only one end of the sequence.
- A stack is called a last-in-first-out (LIFO) collection. This mean that the last thing we added (pushed) is the first thing that gets pulled (popped) off.
- The accessible element of the stack is called the top element.
- The two approaches: array-based and linked stacks
- There are basically three operations that can be performed on stacks .

  (1) inserting an item into a stack (push).

  (2) deleting an item from the stack (pop).

  (3) displaying the contents of the stack(pip).

The C++ ADT for a Stack

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}               // Protect copy constructor

public:
  Stack() {}                           // Default constructor
  virtual ~Stack() {}                  // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

Stack Specification

- A stack ADT allows the following operations:
  - ✓ Push: Add element to top of stack
  - ✓ Pop: Remove element from top of stack
  - ✓ IsEmpty: Check if stack is empty
  - ✓ IsFull: Check if stack is full
  - ✓ Peek: Get the value of the top element without removing it

Use of stack

- The most common uses of a stack are:
  - ✓ **To reverse a word** - Can get the letters in reverse order because of LIFO order of stack
  - ✓ **In programming** - Can check missing elements in the case of parentheses
  - ✓ **In browsers** - Can access visited previously urls

Array Based Stack

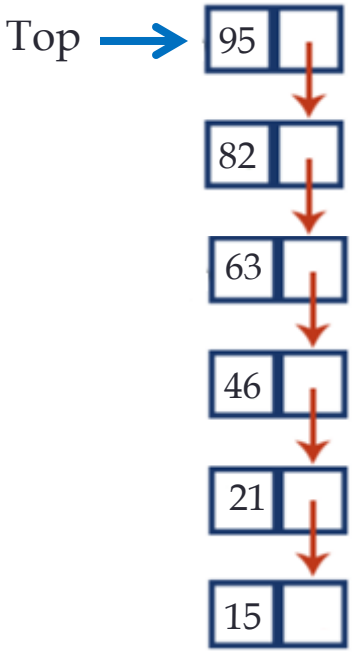
- If the stack's capacity is limited to a certain value, overfilling the stack will cause an error.
- In spite of capacity limitation, array-based implementation is widely applied.
- In a number of cases, required stack capacity is known in advance
- Implementation
  - ✓ empty array
  - ✓ field to record where the next data gets placed into
  - ✓ if array is full, push() returns false,
    otherwise adds it into the correct spot
  - ✓ if array is empty, pop() returns null,
    otherwise removes the next item in the stack

Linked Stack

- The major problem with the stack implemented using array
  - ✓ fixed number of data
  - ✓ need to specify the amount of data in advance
  - ✓ not suitable for the unknown data size
- One disadvantage of a stack implemented by using array is the wasted space.
- The advantage of stack implemented using linked list:
  - ✓ variable size of data
  - ✓ no need to fix the size
  - ✓ as many data values as we want

Linked Stack (cont.)

- Every new element is inserted as '**top**' element pointed by '**top**'.
- To delete an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list.
- The **next** field of the first element : **NULL**.



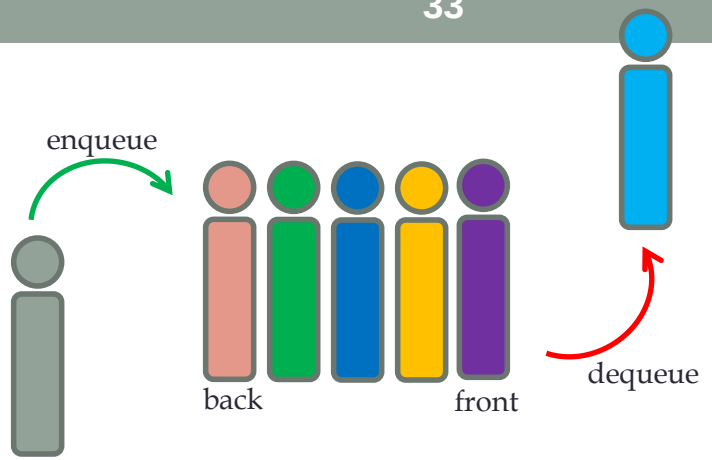
Queue

- Like the stack, the queue is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back and removed from the front.
- Queues operate like standing in line at a movie theater ticket counter.1 If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival.
- Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first too.
- Putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".
- Two implementations for queues: the array-based queue and the linked queue

The C++ ADT for a Queue

```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}      // Protect assignment
  Queue(const Queue&) {}                // Protect copy constructor

public:
  Queue() {}                 // Default
  virtual ~Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&    it) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

Queue Specifications



- Enqueue: Add/insert element to end of queue
- Dequeue: Remove/delete element from front of queue
- IsEmpty: Check if queue is empty or not
- IsFull: Check if queue is full or not

Queue Applications

- people on an escalator
- Waiting ticket outside the cinema
- Cashier line in supermarket
- cars in toll gate

Dictionaries

- The most common objective of computer programs is to store and retrieve data.
- The data structure can be implemented efficient ways to organize collections of data records so that they can be stored and retrieved quickly.
- A simple interface is described for such a collection, called a dictionary. The dictionary ADT provides operations for storing records, finding records, and removing records from the collection.
- At first, define the concepts of a key and comparable objects. If we want to search for a given record in a database, how should we describe what we are looking for?
- A database record could simply be a number, or it could be quite complicated, such as a payroll record with many fields of varying types.

Dictionaries (cont.)

- For example, if searching for payroll records, we might wish to search for the record that matches a particular ID number. In this example the ID number is the search key.
- To implement the search function, we require that keys be comparable.

A payroll record implementation

```
// A simple payroll entry with ID, name, address fields
class Payroll {
private:
  int ID;
  string name;
  string address;

public:
  // Constructor
  Payroll(int inID, string inname, string inaddr) {
    ID = inID;
    name = inname;
    address = inaddr;
  }

  ~Payroll() {}  // Destructor

  // Local data member access functions
  int getID() { return ID; }
  string getname() { return name; }
  string getaddr() { return address; }
};
```

Next Week Lecture (Week 5)

Lecture 5: Binary Trees

Thank you!