CMSC 341
Lecture 7 – STL Containers, Iterators, Algorithms

Based on slides by Jeremy Dixon & Tamassia Goodrich at UMBC

Today's Topics

- Quick Review of: Stacks & Queues
- STL Stacks & Queues
- Deque
- Vectors
- Lists
- Iterators
- Sequences
- Bonus

Standard Template Library (STL)

Standard Template Library (STL)

- The Standard Template Library (*STL*) is a C++ library of container classes, algorithms, and iterators
- Provides many of the basic algorithms and data structures of computer science

Considerations of the STL

- The decision of which type of container to use for a specific need depends on:
  - the functionality offered by the container
  - the efficiency of some of its members (complexity)

From: http://www.cplusplus.com/reference/stl/

Types of Containers

Focus of Today

- Sequence containers
  - Array, vector, deque, list, forward_list
- Container adapters
  - Stacks, queues, priority_queues
- Associative containers (and the unordered)
  - Set, multiset, map, multimap

Standard Containers

- Sequences:
  - **vector**: Dynamic array of variables, struct or objects. Insert data at the end.
  - **list**: Linked list of variables, struct or objects. Insert/remove anywhere.
  - Sequence **means order does matter**

Container Adapters

- Container adapters:
  - **stack** LIFO
  - **queue** FIFO
  - adapter means ***VERY LIMITED*** functionality

Will we use STL?

- Today we are going to talk about the ways that we can implement stacks, queues, deque, vector, list, iterators, algorithms.

- Review: 3 Ways to Create a Stack or Queue
  - Create a static stack or queue using an array
  - Create a dynamic stack or queue using a linked list
  - Create a stack or queue using the STL

Stacks

Implementations of Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list
- Using STL (dynamic)

Stack Operations

- Push
  - causes a value to be stored in (pushed onto) the stack
- Pop
  - retrieves and removes a value from the stack

Other Stack Operations

- **`isFull()`**: A Boolean operation needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- **`isEmpty()`**: A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.

Static Stacks

Static Stacks

- A *static stack* is built on an array
  - As we are using an array, we must specify the starting size of the stack
  - The stack may become full if the array becomes full

Member Variables for Stacks

- Three major variables:
  - **`Pointer`** Creates a pointer to stack
  - **`size`** Tracks elements in stack
  - **`top`** Tracks top element in stack

Member Functions for Stacks

– **`CONSTRUCTOR`**	Creates a stack
– **`DESTRUCTOR`**	Deletes a stack
– **`push()`**	Pushes element to stack
– **`pop()`**	Pops element from stack
– **`isEmpty()`**	Is the stack empty?
– **`isFull()`**	Is the stack full?

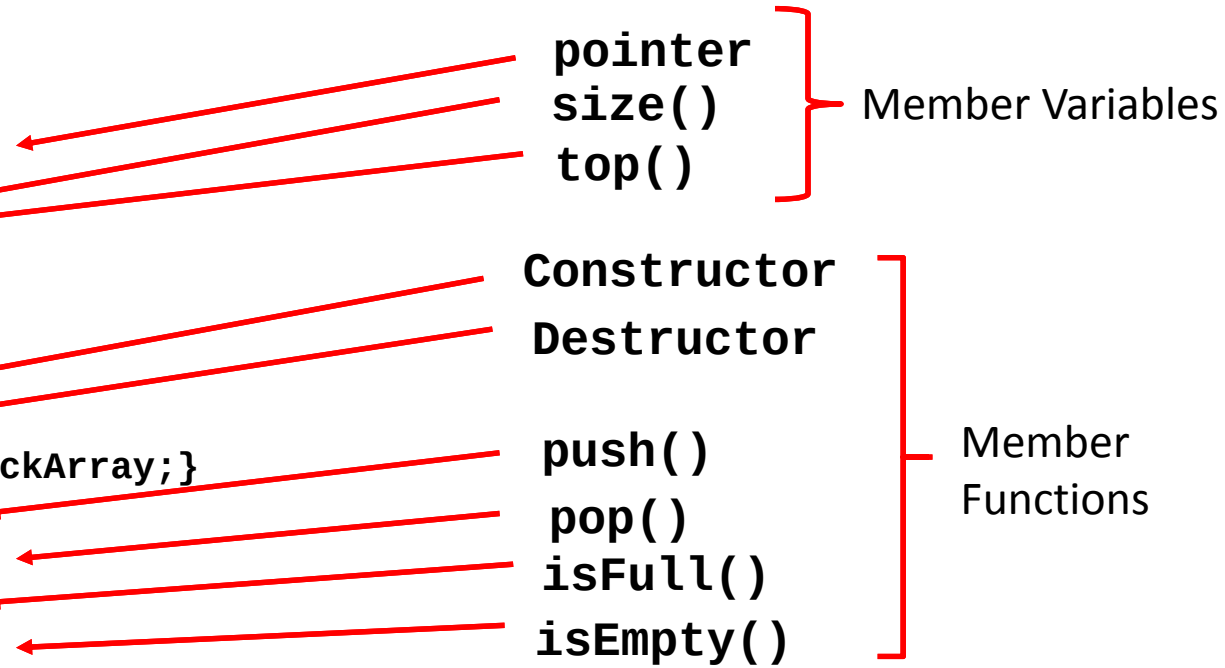
Static Stack Definition

```
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
        int *stackArray;
        int stackSize;
        int top;

public:
        IntStack(int);
        ~IntStack()
           {delete[] stackArray;}
        void push(int);
        void pop(int &);
        bool isFull();
        bool isEmpty();
};

#endif
```



Dynamic Stacks

Dynamic Stacks

- A *dynamic stack* is built on a linked list instead of an array.
- A linked list-based stack offers two advantages over an array-based stack.
  - No need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, and then expands by one node each time a value is pushed.
  - A dynamic stack will never be full, as long as the system has enough free memory.

Member Variables for Dynamic Stacks

- Parts:
  - **`Linked list`** Linked list for stack (nodes)
  - **`size`** Tracks elements in stack

Member Functions for Dynamic Stacks

– **`CONSTRUCTOR`**	Creates a stack
– **`DESTRUCTOR`**	Deletes a stack
– **`push()`**	Pushes element to stack
– **`pop()`**	Pops element from stack
– **`isEmpty()`**	Is the stack empty?
– **`top()`**	What is the top element?

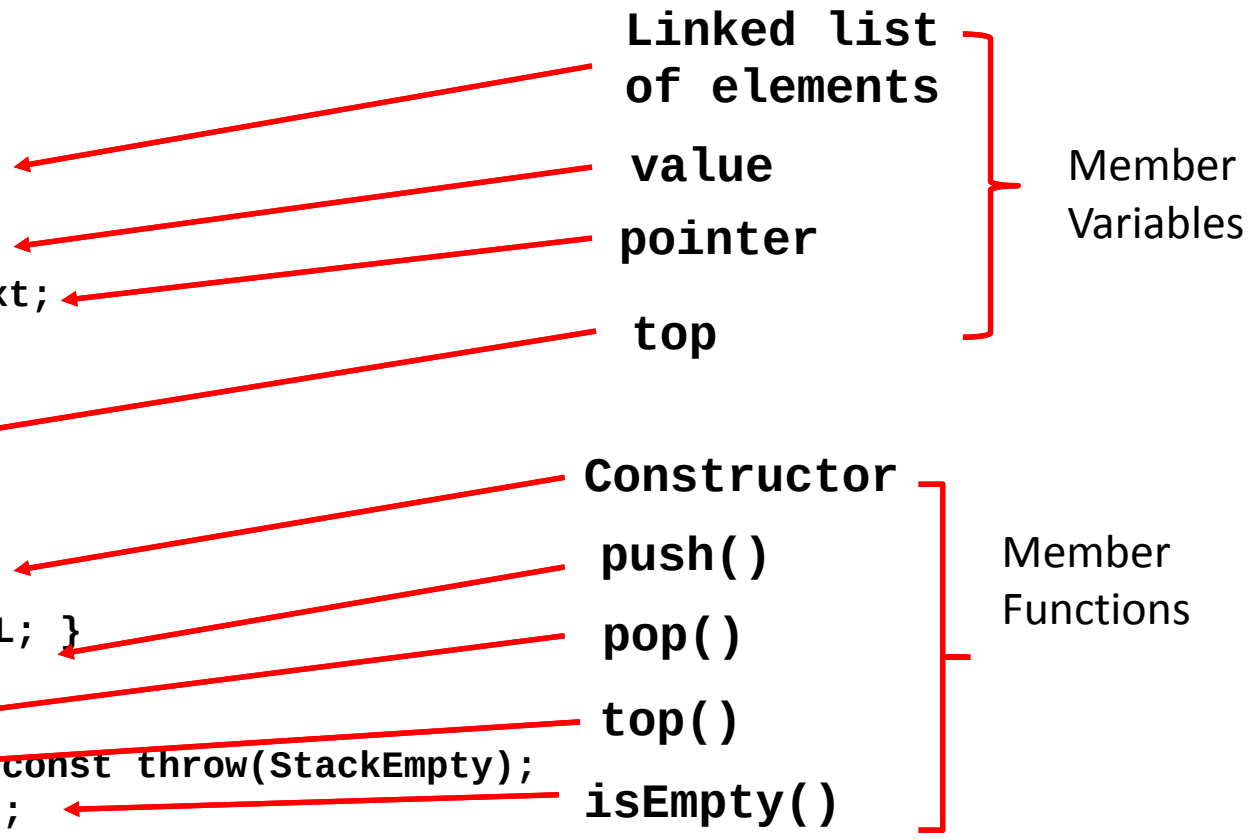
What happened to `isFull()`?

Dynamic Stack

```
class DynIntStack
{
private:
     struct StackNode
     {
          int value;
          StackNode *next;
     };

     StackNode *top;

public:
     DynIntStack(void)
          {   top = NULL; }
     void push(int);
     void pop(int &);
     const Elem& top() const throw(StackEmpty);
     bool isEmpty(void);
};
```



The STL Stack

```
#include <stack>
using std::stack; // make stack accessible
stack<int> myStack; // a stack of integers
```

List of the principal member functions.

- size(): Return the number of elements in the stack.
- empty(): Return true if the stack is empty and false otherwise. push(e): Push e onto the top of the stack.
- pop(): Pop the element at the top of the stack.
- top(): Return a reference to the element at the top of the stack

Common Problems with Stacks

- Stack underflow
  - no elements in the stack, and you tried to pop
- Stack overflow
  - maximum elements in stack, and tried to add another
  - not an issue using STL or a dynamic implementation

Queues

Introduction to the Queue

- Like a stack, a queue is a data structure that holds a sequence of elements.
- A queue, however, provides access to its elements in *first-in, first-out (FIFO)* order.

Implementations of Queues

- Static Queues
  - Fixed size
  - Can be implemented with an array
- Dynamic Queues
  - Grow in size as needed
  - Can be implemented with a linked list
- Using STL (dynamic)

Just like stacks!

Implementation of a Static Queue

- The previous discussion was about static arrays
  - Container is an array
- Class Implementation for a static integer queue
  - Member functions
    - **`enqueue()`**
    - **`dequeue()`**
    - **`isEmpty()`**
    - **`isFull()`**
    - **`clear()`**

Member Variables for Static Queues

- Five major variables:
  - **`queueArray`** Creates a pointer to queue
  - **`queueSize`** Tracks capacity of queue
  - **`numItems`** Tracks elements in queue
  - **`front`**
  - **`rear`**
    - The variables front and rear are used when our queue "rotates," as discussed earlier

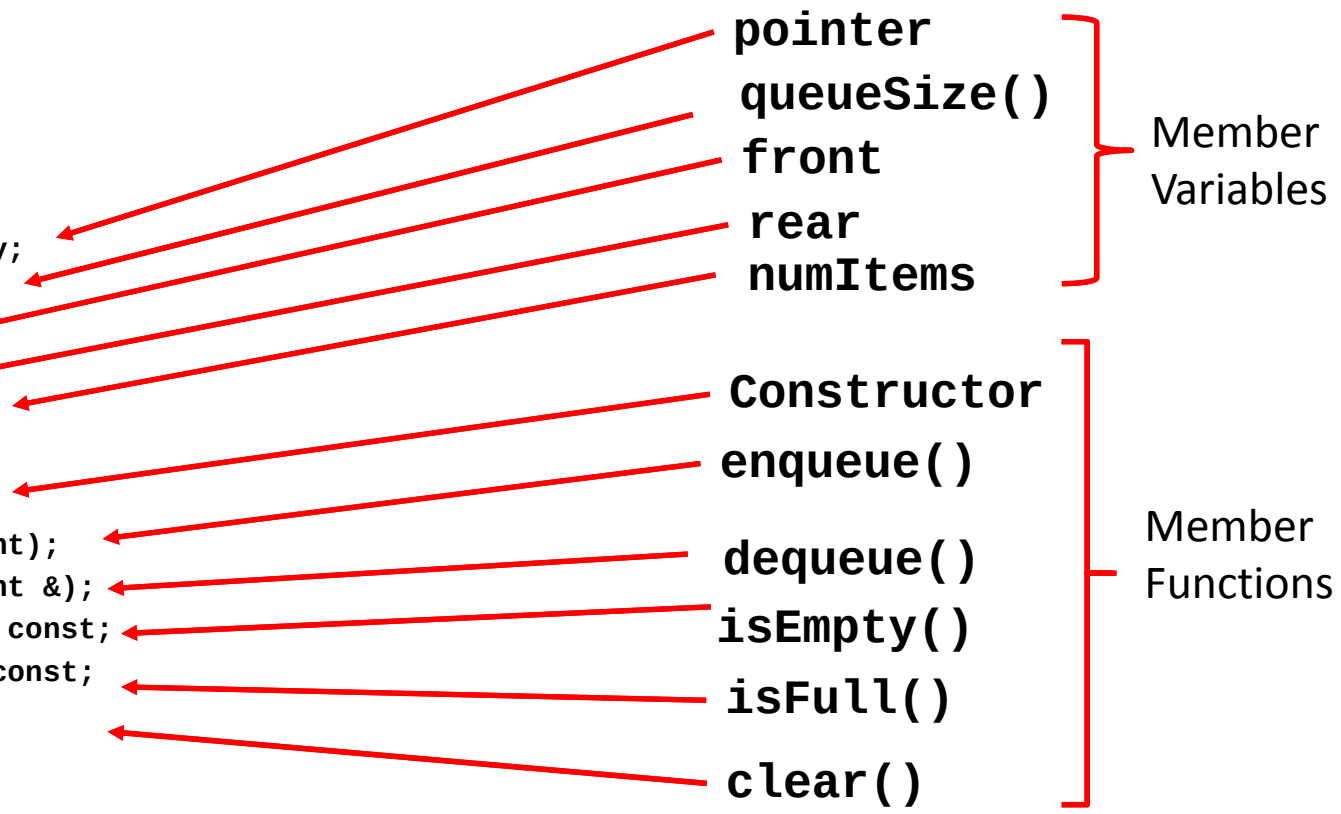
Member Functions for Queues

- **`CONSTRUCTOR`** Creates a queue
- **`DESTRUCTOR`** Deletes a queue
- **`enqueue()`** Adds element to queue
- **`dequeue()`** Removes element from queue
- **`isEmpty()`** Is the queue empty?
- **`isFull()`** Is the queue full?
- **`clear()`** Empties queue

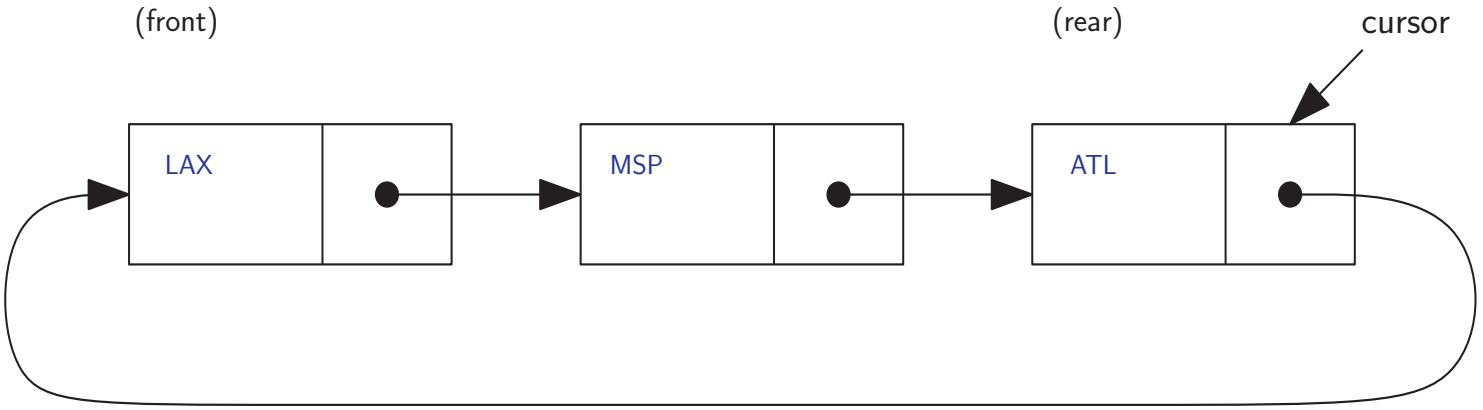
Static Queue Example

```
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
      int *queueArray;
      int queueSize;
      int front;
      int rear;
      int numItems;
public:
      IntQueue(int);
      void enqueue(int);
      void dequeue(int &);
      bool isEmpty() const;
      bool isFull() const;
      void clear();
};
#endif
```



Dynamic Queue Example



Circularly Linked List

STL Queues

STL Queues

- Another way to implement a queue is by using the standard library
- An STL queue leverages the pre-existing library to access the data structure
- Principal member functions:
  - size(): Return the number of elements in the queue.
  - empty(): Return true if the queue is empty and false otherwise.
  - push(e): Enqueue e at the rear of the queue.
  - pop(): Dequeue the element at the front of the queue.
  - front(): Return a reference to the element at the queue's front.
  - back(): Return a reference to the element at the queue's rear.

STL Queue Example

```
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue
using namespace std;

int main ()
{
  std::queue<int> myqueue;
  int myint;

  std::cout << "Please enter some integers (enter 0 to
end):\n";

  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);

  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
```

STL Deque

Deque - Double ended Queue (Implement it with a doubly Linked List)
(Supports insertion and deletion at both the front and the rear of the queue)

Here is a list of the principal operations.

- size(): Return the number of elements in the deque.
- empty(): Return true if the deque is empty and false otherwise.
- push front(e): Insert e at the beginning the deque.
- push back(e): Insert e at the end of the deque.
- pop front(): Remove the first element of the deque.
- pop back(): Remove the last element of the deque.
- front(): Return a reference to the deque's first element.
- back(): Return a reference to the deque's last element.

Warm up Question!!

Explain how you can implement all the functions of the deque ADT using two stacks. What is the running time of the two stacks deque functions?

- size(): Return the number of elements in the deque.
- empty(): Return true if the deque is empty and false otherwise.
- push front(e): Insert e at the beginning the deque.
- push back(e): Insert e at the end of the deque.
- pop front(): Remove the first element of the deque.
- pop back(): Remove the last element of the deque.
- front(): Return a reference to the deque's first element.
- back(): Return a reference to the deque's last element.

Bonus Question (0.5%)

Describe how to implement the stack ADT using two queues. What is the running time of the push and pop functions in this case?

A stack is an abstract data type (ADT) that supports the following operations:
push(e): Insert element e at the top of the stack.
pop(): Remove the top element from the stack; an error occurs if the stack is empty.
top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:
size(): Return the number of elements in the stack.
empty(): Return true if the stack is empty and false otherwise.

Iterators

Containers and Iterators

- An iterator abstracts the process of scanning through a collection of elements
- A container is an abstract data structure that supports element access through iterators
  - begin(): returns an iterator to the first element
  - end(): return an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element
  - *p: returns the element referenced by this iterator
  - ++p: advances to the next element
- Extends the concept of position by adding a traversal capability

Containers

- Data structures that support iterators are called containers
- Examples include Stack, Queue, Vector, List
- Various notions of iterator:
  - (standard) iterator: allows read-write access to elements
  - const iterator: provides read-only access to elements
  - bidirectional iterator: supports both ++p and –p
  - random-access iterator: supports both p+i and p-i

Iterating through a Container

- Let C be a container and p be an iterator for C

```
for (p = C.begin(); p != C.end(); ++p)
    loop_body
```

- Example: (with an STL vector)

```
typedef vector<int>::iterator Iterator;
int sum = 0;
for (Iterator p = V.begin(); p != V.end(); ++p)
sum += *p;
return sum;
```

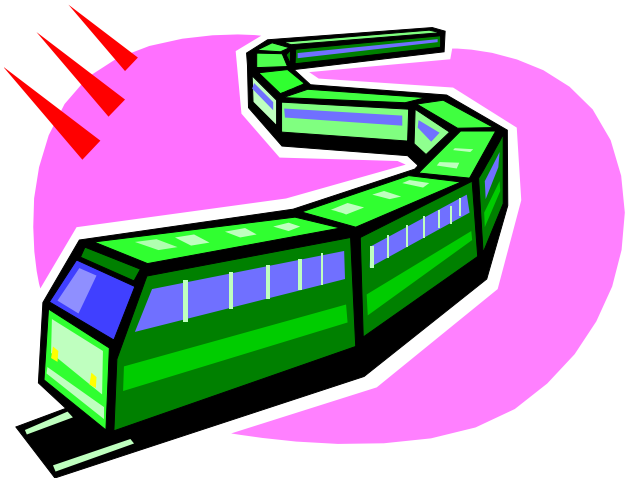
Implementing Iterators

- Array-based
  - array A of the n elements
  - index i that keeps track of the cursor
  - begin() = 0
  - end() = n (index following the last element)
- Linked list-based
  - doubly-linked list L storing the elements, with sentinels for header and trailer
  - pointer to node containing the current element
  - begin() = front node
  - end() = trailer node (just after last node)

STL Iterators in C++

- Each STL container type C supports iterators:
  - C::iterator – read/write iterator type
  - C::const_iterator – read-only iterator type
  - C.begin(), C.end() – return start/end iterators
- This iterator-based operators and methods:
  - *p: access current element
  - ++p, --p: advance to next/previous element
  - C.assign(p, q): replace C with contents referenced by the iterator range [p, q) (from p up to, but not including, q)
  - insert(p, e): insert e prior to position p
  - erase(p): remove element at position p
  - erase(p, q): remove elements in the iterator range [p, q)

Lists



Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object p.element(): returns the element at position
  - In C++ it is convenient to implement this as *p

Node List ADT

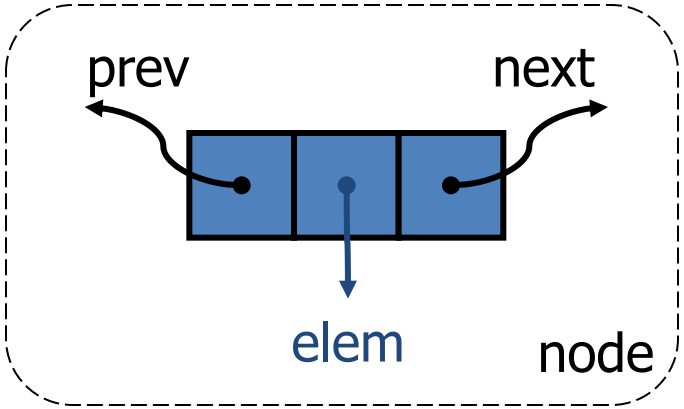
- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), empty()
- Iterators:
  - begin(), end()
- Update methods:
  - insertFront(e), insertBack(e)
  - removeFront(), removeBack()
- Iterator-based update:
  - insert(p, e)
  - remove(p)

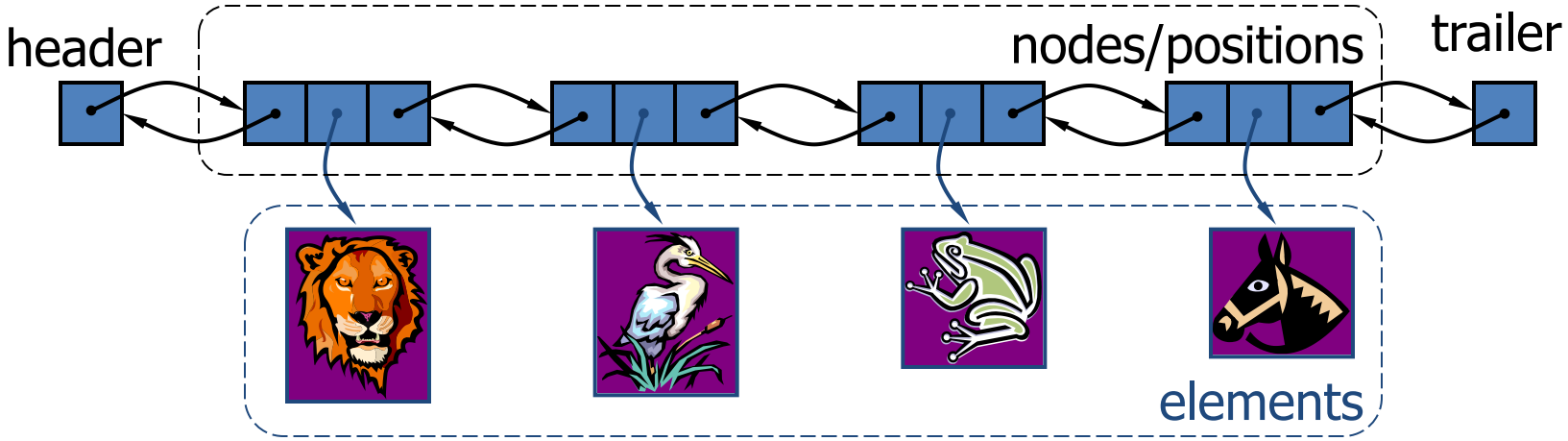
Node List ADT

Operation	*Output*	*L*
insertFront(8)	–	(8)
$p =$ begin()	p : (8)	(8)
insertBack(5)	–	(8,5)
$q = p;\ ++q$	q : (5)	(8,5)
$p ==$ begin()	*true*	(8,5)
insert(q,3)	–	(8,3,5)
$*q = 7$	–	(8,3,7)
insertFront(9)	–	(9,8,3,7)
eraseBack()	–	(9,8,3)
erase(p)	–	(9,3)
eraseFront()	–	(3)

Doubly Linked List

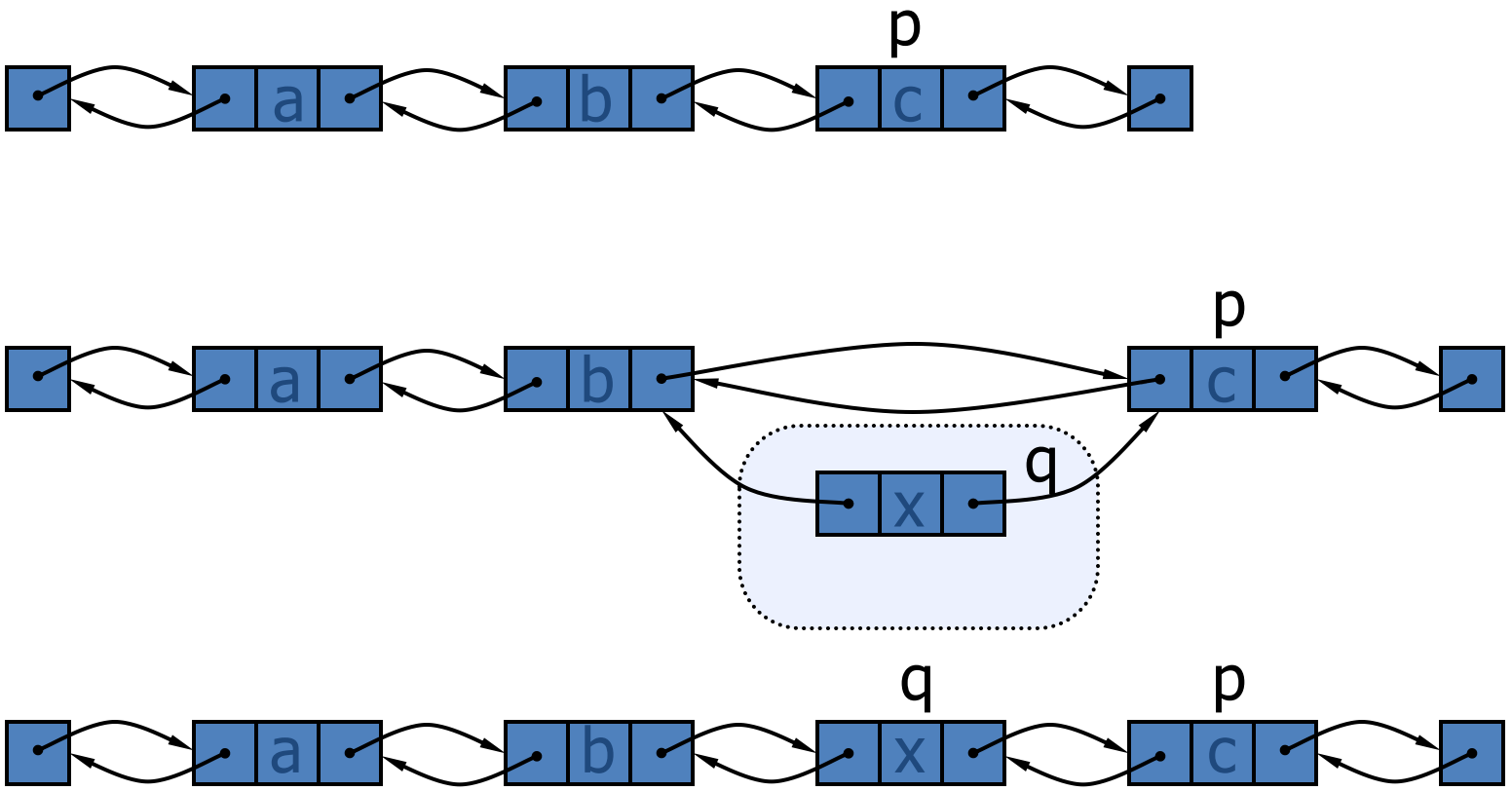
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insert(p, x), which inserts x before p



Insertion Algorithm

Algorithm insert(p, e): {insert e before p}

Create a new node v

v→element = e

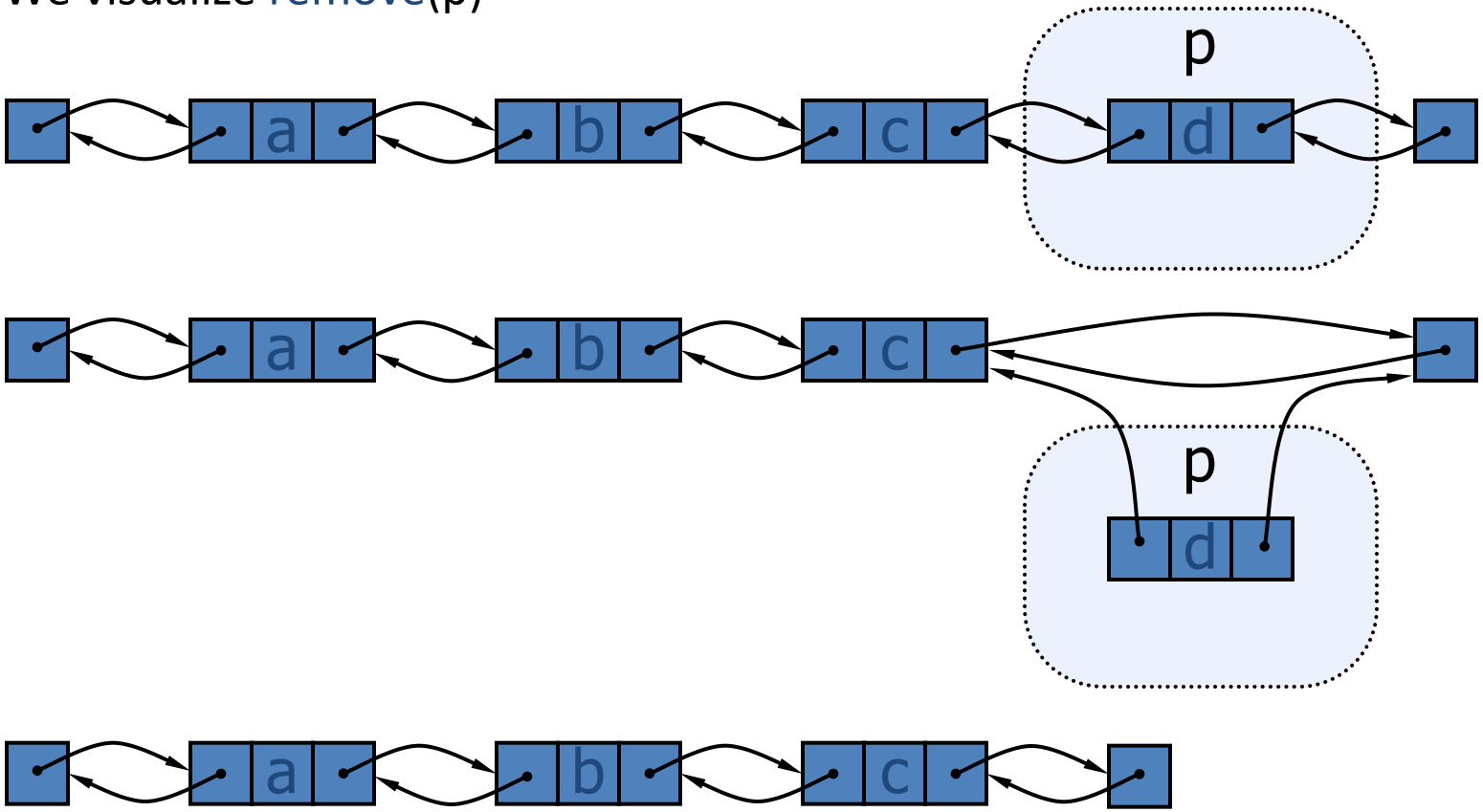
u = p→prev

v→next = p; p→prev = v {link in v before p}

v→prev = u; u→next = v {link in v after u}

Deletion

- We visualize remove(p)



Deletion Algorithm

Algorithm remove(p):

u = p→prev

w = p→next

u→next = w {linking out p}

w→prev = u

Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

Sequence ADT

- The Sequence ADT is the union of the Array List and Node List ADTs
- Elements accessed by
  - Index, or
  - Position
- Generic methods:
  - size(), empty()
- Vector (ArrayList)-based methods:
  - at(i), set(i, o), insert(i, o), erase(i)
- List-based methods:
  - begin(), end()
  - insertFront(o), insertBack(o)
  - eraseFront(), eraseBack()
  - insert (p, o), erase(p)
- Bridge methods:
  - atIndex(i), indexOf(p)

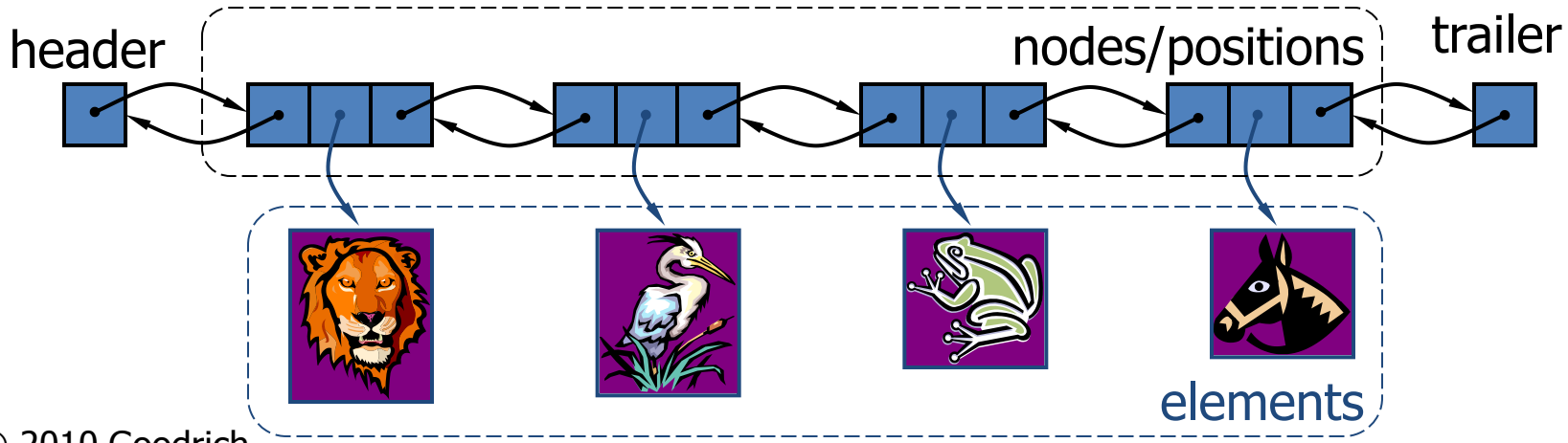
Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

Linked List Implementation

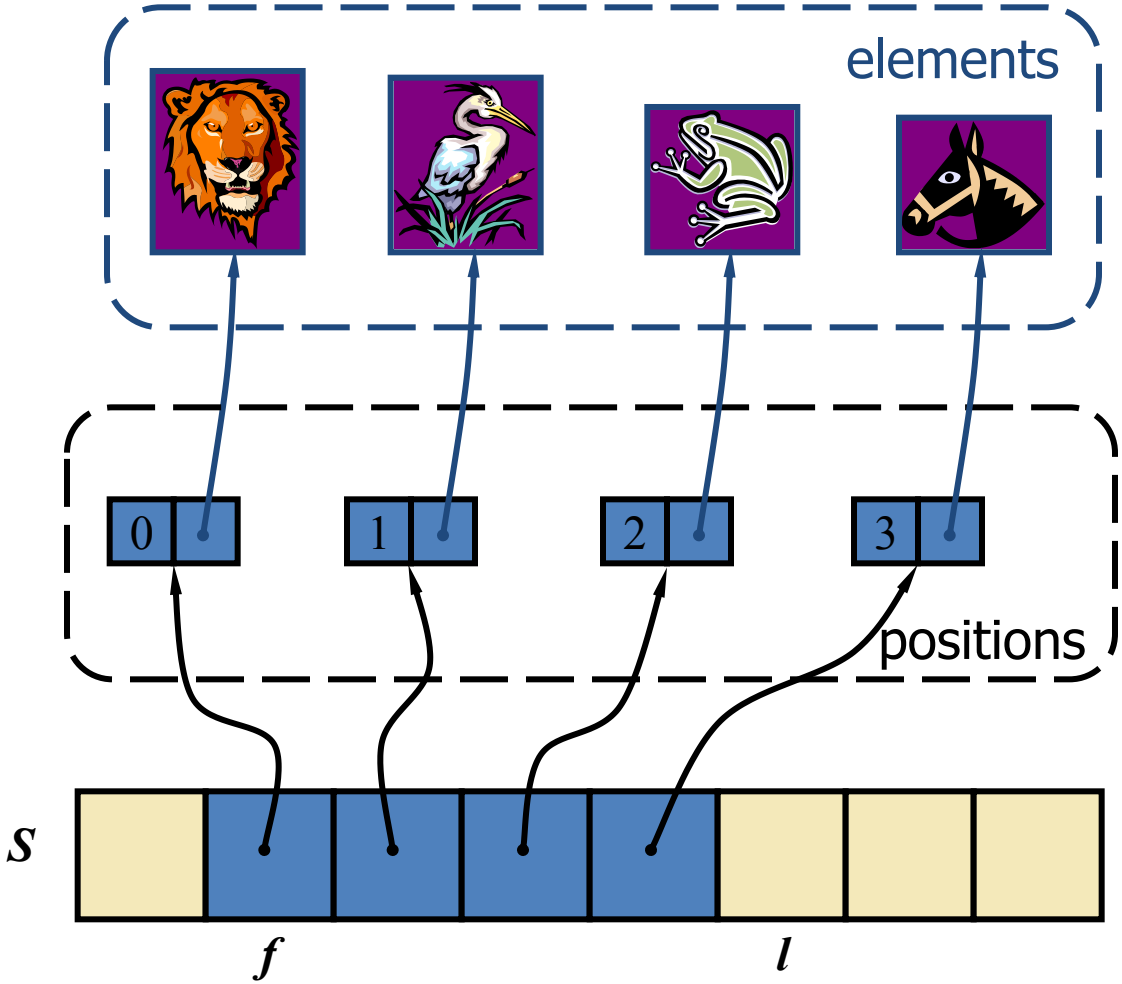
- A doubly linked list provides a reasonable implementation of the Sequence ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

- Position-based methods run in constant time
- Index-based methods require searching from header or trailer while keeping track of indices; hence, run in linear time



Circular Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Index
- Indices f and l keep track of first and last positions



Comparing Sequence Implementations

Operations	*Circular Array*	*List*
size, empty	$O(1)$	$O(1)$
atIndex, indexOf	$O(1)$	$O(n)$
begin, end	$O(1)$	$O(1)$
$*p, ++p, --p$	$O(1)$	$O(1)$
insertFront, insertBack	$O(1)$	$O(1)$
insert, erase	$O(n)$	$O(1)$