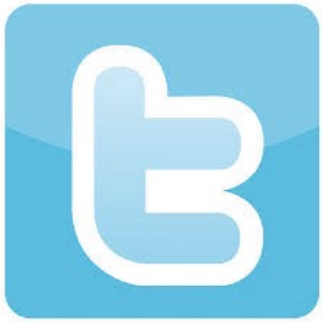
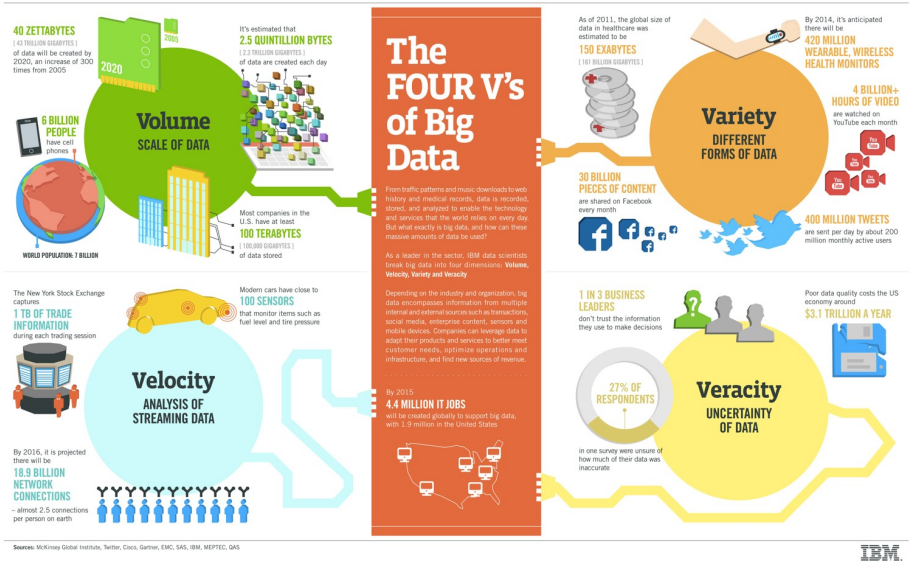
Introduction to Basic Data Structures

Stacks and queues

Modern world all about … **DATA**

Google™





Big data

From Wikipedia, the free encyclopedia

This article is about large collections of data. For the graph database, see Graph database. For the band, see Big Data (band).

Big data is an all-encompassing term for any collection of data sets so large and complex that it becomes difficult to process using on-hand data management tools or traditional data processing applications.

The challenges include capture, curation, storage, search, sharing, transfer, analysis and visualization. The trend to larger data sets is due to the additional information derivable from analysis of a single large set of related data, as compared to separate smaller sets with the same total amount of data, allowing correlations to be found to "spot business trends, prevent diseases, combat crime and so on."[1]

Scientists regularly encounter limitations due to large data sets in many areas, including meteorology, genomics,[2] connectomics, complex physics simulations,[3] and biological and environmental research.[4] The limitations also affect Internet search, finance and business informatics. Data sets grow in size in part because they are increasingly being gathered by ubiquitous information-sensing mobile devices, aerial sensory technologies (remote sensing), software logs, cameras, microphones, radio-frequency identification (RFID) readers, and wireless sensor networks.[5][6][7] The world's technological per-capita capacity to store information has roughly doubled every 40 months since the 1980s;[8] as of 2012, every day 2.5 exabytes (2.5×10^{18}) of data were created.[9] The challenge for large enterprises is determining who should own big data initiatives that straddle the entire organization.[10]

A visualization created by IBM of Wikipedia edits. At multiple terabytes in size, the text and images of Wikipedia are a classic example of big data.

Big data is difficult to work with using most relational database management systems and desktop statistics and visualization packages, requiring instead "massively parallel software running on tens, hundreds, or even thousands of servers".[11] What is considered "big data" varies depending on the

Dealing with data…

- How to use it ?
- How to store it ?
- How to process it ?
- How to gain “knowledge” from it ?
- How to keep it secret?

Dealing with data…

- How to use it ?
- How to store it ?
- How to process it ?
- How to gain “knowledge” from it ?
- How to keep it secret?

How should data be stored?

Depends on your requirement

"We back up our data on sticky notes because sticky notes never crash."

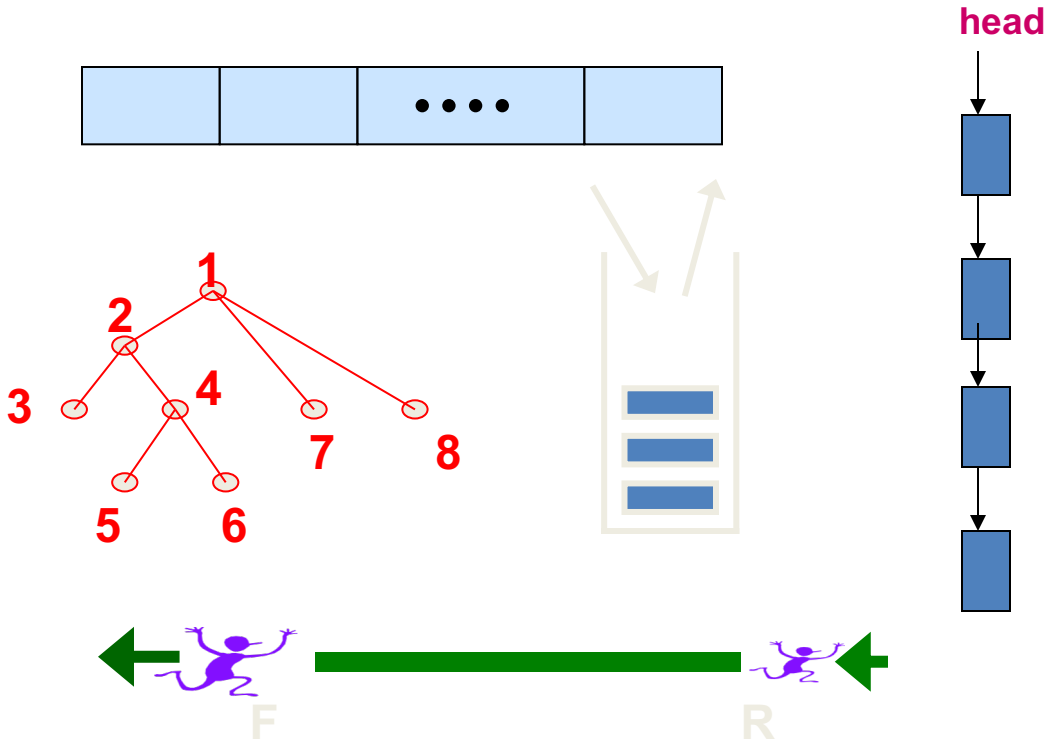
Data is diverse ..

But we have some building blocks

To store our big data

Elementary Data "Structures"

- **Arrays**
- **Lists**
- **Stacks**
- **Queues**
- **Trees**



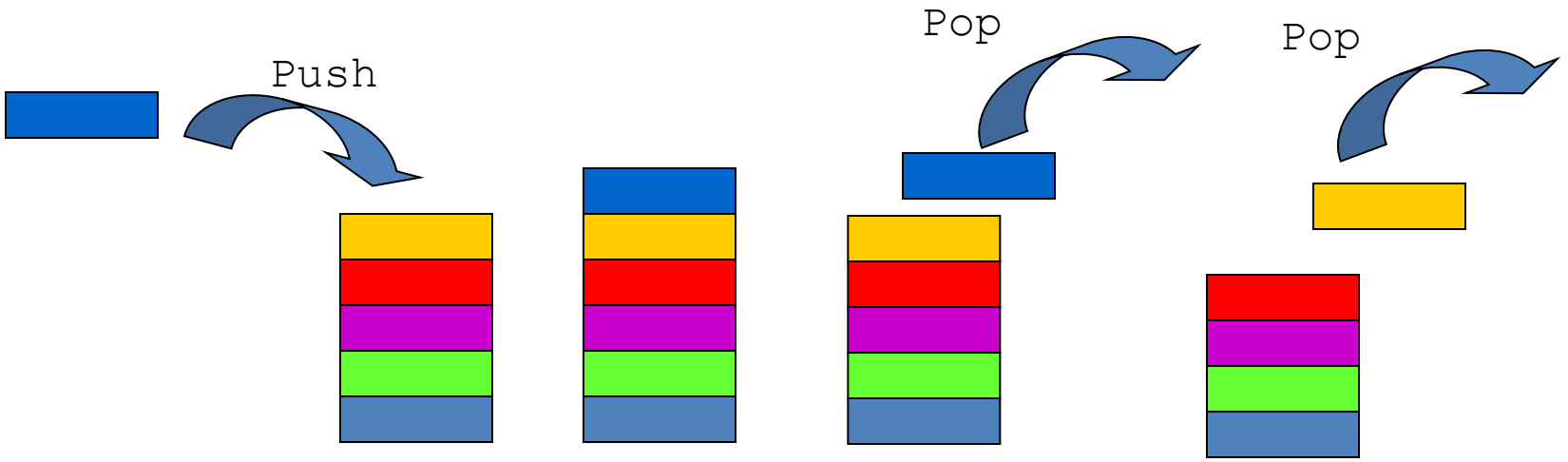
In some languages these are basic data types – in others they need to be implemented

Stacks

Stack

A list for which Insert and Delete are allowed only at one end of the list (the *top*)

– LIFO – Last in, First out



What is this good for ?

- Page-visited history in a Web browser

What is this good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor

What is this good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Saving local variables when one function calls another, and this one calls another

How should we represent it ?

- Write code in python ?

How should we represent it ?

- Write code in python ?
- Write code in C ?

How should we represent it ?

- Write code in python ?
- Write code in C ?
- Write code in Java ?

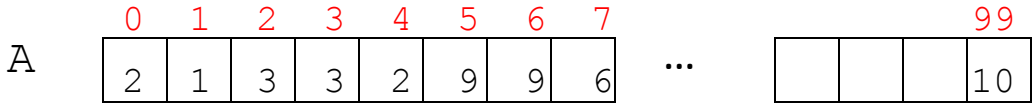
Aren’t we essentially doing the same thing?

Abstract Data Type

A mathematical definition of objects, with operations defined on them

Examples

- Basic Types
  - integer, real (floating point), boolean (0,1), character
- Arrays
  - A[0..99] : integer array



  - A[0..99] : array of images

ADT: Array

A mapping from an index set, such as `{0,1,2,…,n}`, into a cell type

Objects: set of cells

Operations:

- `create(A,n)`
- `put(A,v,i)` or `A[i] = v`
- `value(A,i)`

Also the "general" definition for functions

Abstract Data Types (ADTs)

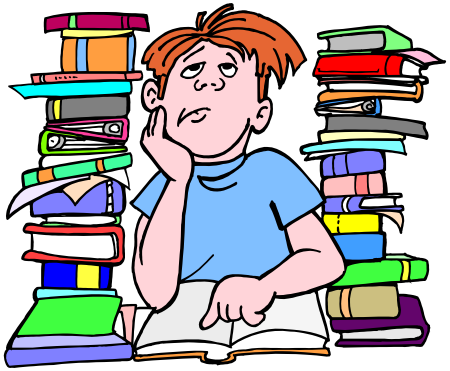
- An abstract data type (ADT) is an abstraction of a data structure

- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

ADT for stock trade

- The data stored are buy/sell orders
- The operations supported are
  - order buy (stock, shares)
  - order sell(stock, shares)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

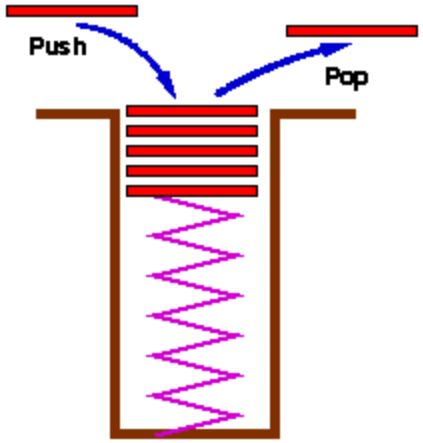
Stack ADT



Objects:

A finite sequence of nodes

Operations:



- Create
- Push: Insert element at top
- Top: Return top element
- Pop: Remove and return top element
- IsEmpty: test for emptyness

Exceptions

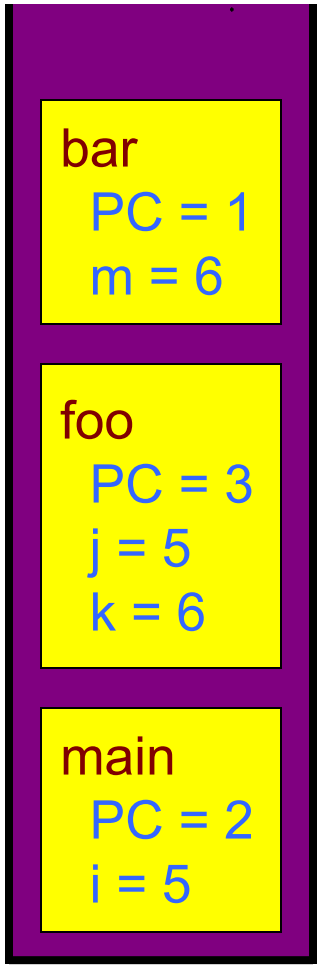
- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Exercise: Stacks

- Describe the output of the following series of stack operations
  - Push(8)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(5)
  - Pop()
  - Pop()
  - Push(9)
  - Push(1)

C++ Run-time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack



Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(X,n):
Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in X match
Let S be an empty stack
for i=0 to n-1 **do**
    if $X[i]$ is an opening grouping symbol **then**
        S.push($X[i]$)
    else if $X[i]$ is a closing grouping symbol **then**
        if S.isEmpty() **then**
            return false *{nothing to match with}*
        if S.pop() does not match the type of $X[i]$ **then**
            return false *{wrong type}*
if S.isEmpty() **then**
    return true *{every symbol matched}*
else
    return false *{some symbols were never matched}*

Postfix Evaluator

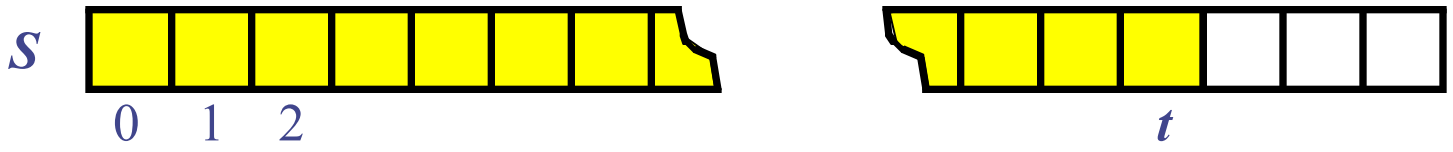
- Postfix: every operator follows all of its operands => no parenthesis required
  - Infix: (3+4)*5
  - Postfix: 3 4 + 5 *
- 5 3 6 * + 7 - = ?
- Write python code to evaluate it

Evaluation is still from left to right

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

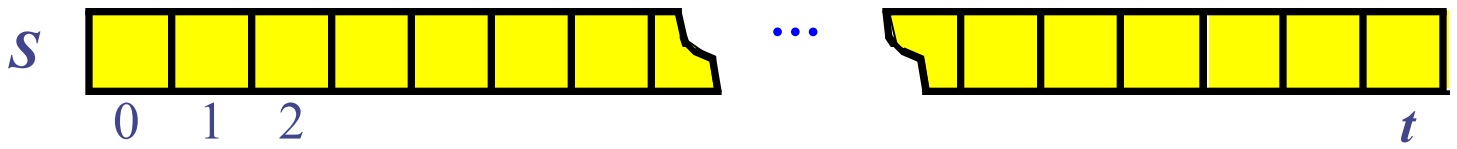
```
Algorithm size()
  return t + 1

Algorithm pop()
  if empty() then
    throw EmptyStackException
   else
    t = t - 1
    return S[t + 1]
```



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
  if t = S.length - 1 then
    throw FullStackException
   else
    t = t + 1
    S[t] = o
```



Performance and Limitations

- array-based implementation of stack ADT

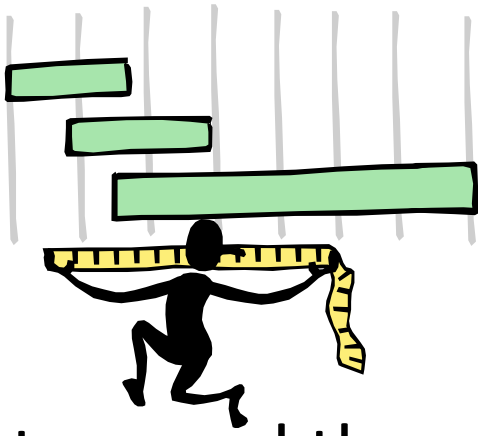
- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori* , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

```
Algorithm push(o)
  if t = S.length - 1
then
    A = new array of
          size …
    for i = 0 to t do
       A[i] = S[i]
       S = A
  t = t + 1
  S[t] = o
```

Comparison of the Strategies



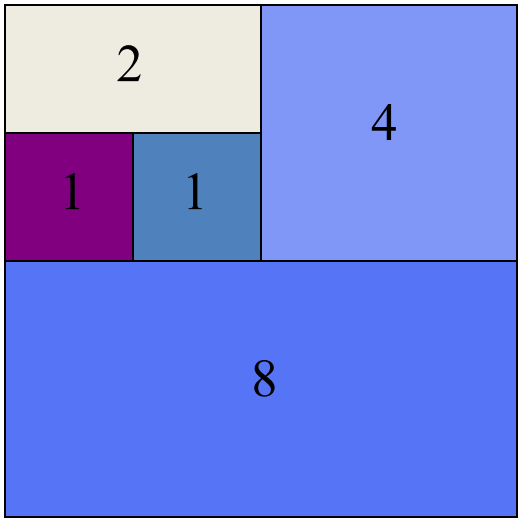
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized time** of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n push operations is proportional to
  - $n + c + 2c + 3c + 4c + \ldots + kc =$
    - $n + c(1 + 2 + 3 + \ldots + k) =$
      - $n + ck(k + 1)/2$
- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of a push operation is $O(n)$

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to
  - $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$
    - $n + 2^{k+1} - 1 = 2n - 1$
- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$



Stack Interface in C++

- Interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Most similar STL construct is vector

```
template <typename Object>
class Stack {
public:
   int size()
   bool isEmpty()
   Object& top()
         throw(EmptyStackException)
   void push(Object o)
   Object pop()
         throw(EmptyStackException);
};
```

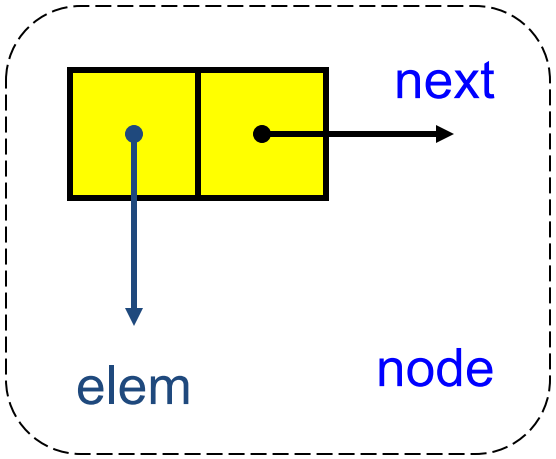
Array-based Stack in C++

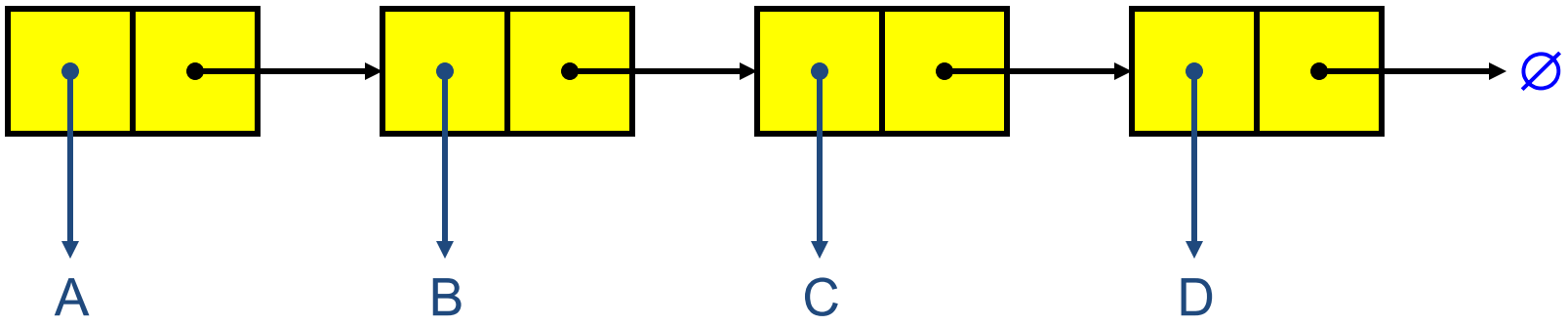
```
template <typename Object>
class ArrayStack {
private:
    int capacity;      // stack capacity
    Object *S;         // stack array
    int top;           // top of stack
public:
    ArrayStack(int c) {
        capacity = c;
        S = new Object[capacity];
        t =  –1;
    }
```

```
        isEmpty()
    {  return (t < 0);  }

          pop()
        {
     if(isEmpty())
        throw EmptyStackException
          ("Access to empty stack");
        return S[t--];
    }
// … (other functions omitted)
```

Singly Linked List

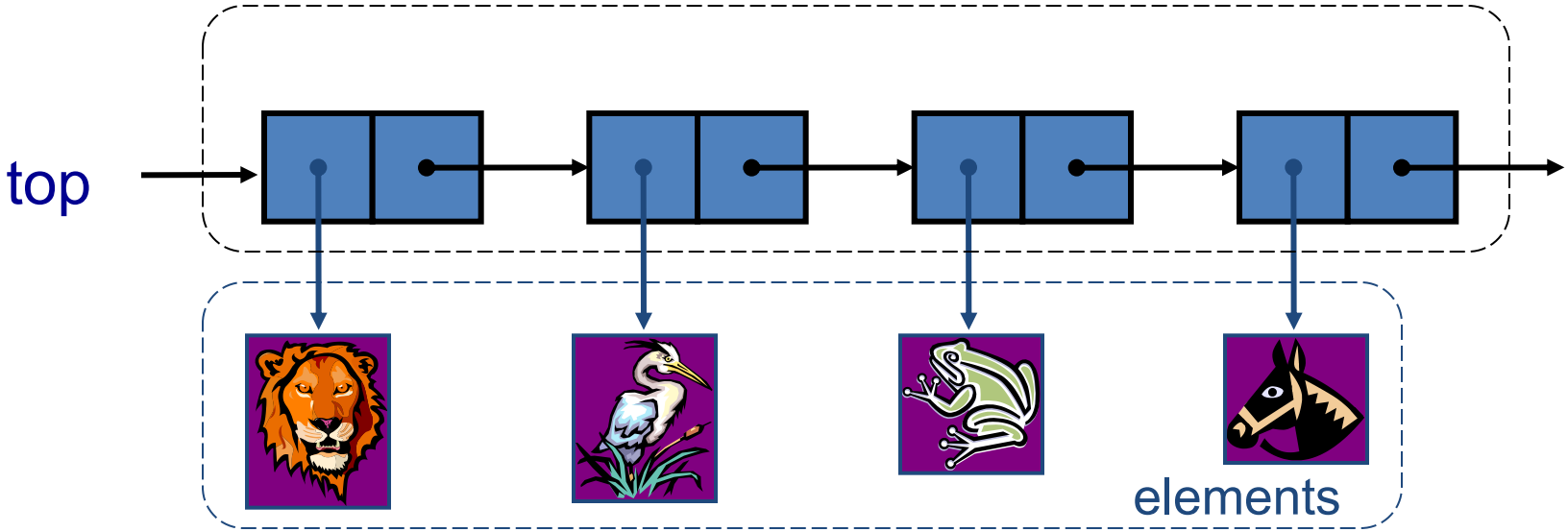
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Exercise

- Describe how to implement the linked list (and its variants) in Python where there is no pointer!

Exercise

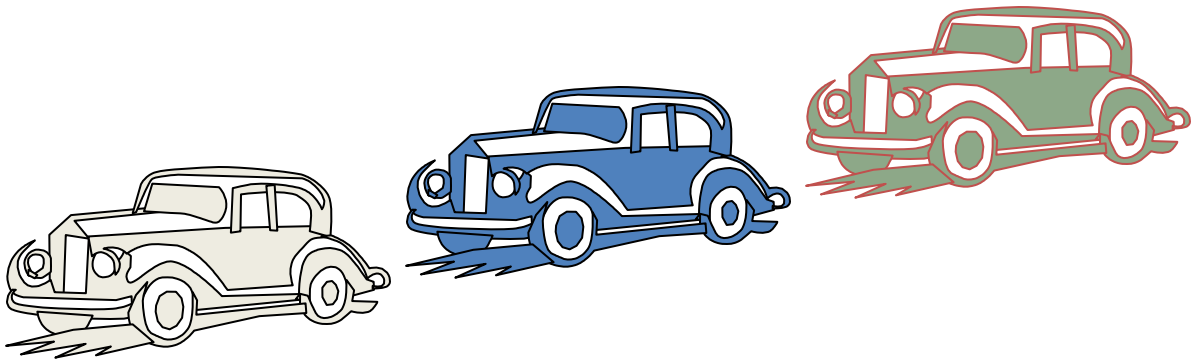
- Describe how to implement a stack using a singly-linked list
  - Stack operations: push(x), pop( ), size(), isEmpty()
  - For each operation, give the running time

Stack Summary

- Stack Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
Pop()	O(1)	O(1)	O(1)
Push(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
Top()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)

Queues



Outline and Reading

- The Queue ADT
- Implementation with a circular array
  - Growable array-based queue
- List-based queue

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object o): inserts element o at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Exercise: Queues

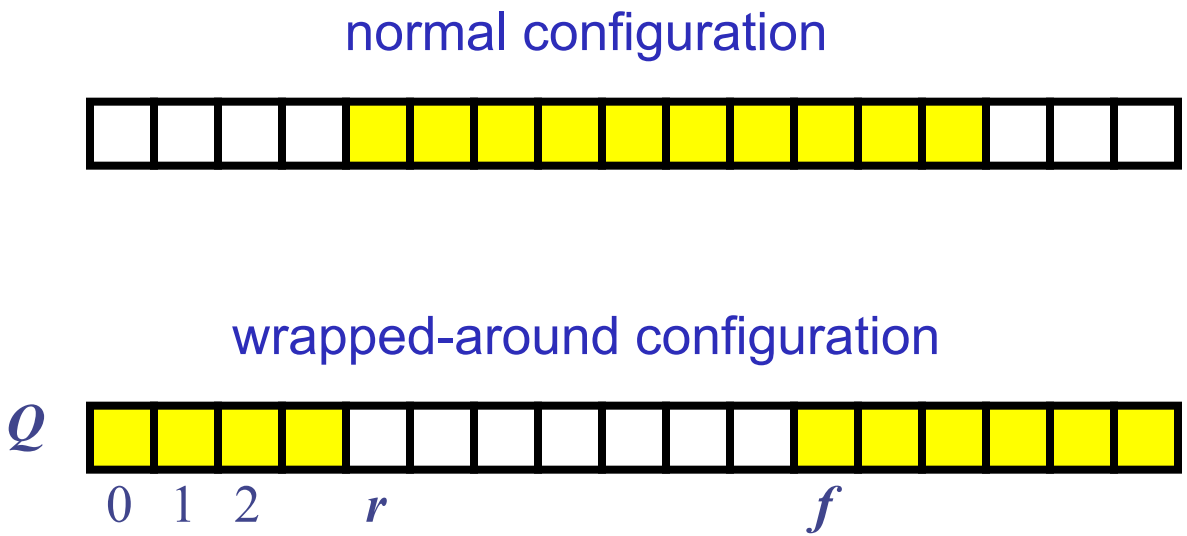
- Describe the output of the following series of queue operations
  - enqueue(8)
  - enqueue(3)
  - dequeue()
  - enqueue(2)
  - enqueue(5)
  - dequeue()
  - dequeue()
  - enqueue(9)
  - enqueue(1)

Applications of Queues

- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

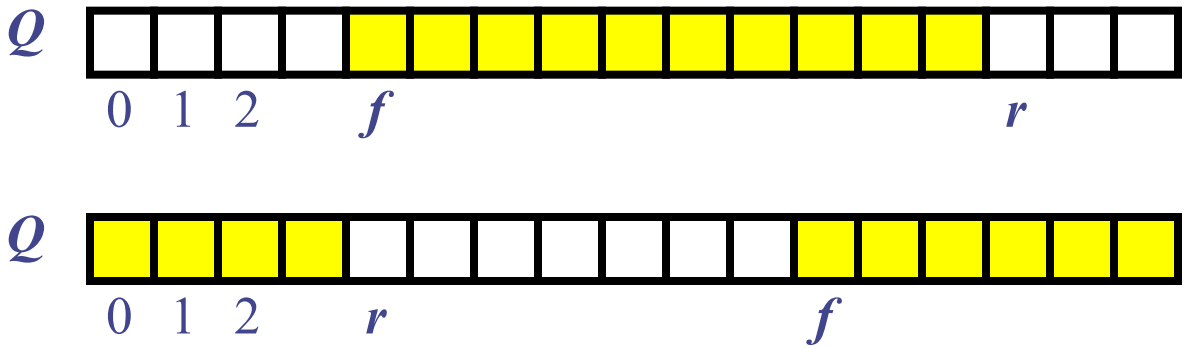
- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty



Queue Operations

- We use the modulo operator (remainder of division)

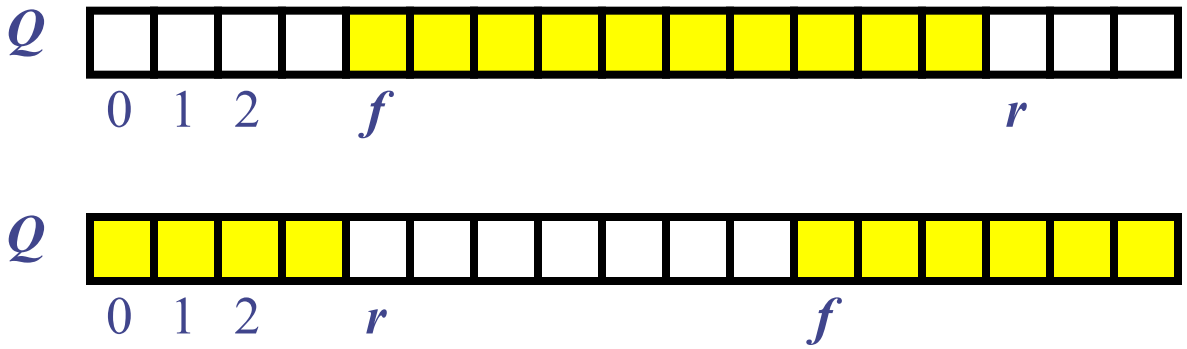
```
Algorithm size()
  return (N + r − f) mod N

Algorithm isEmpty()
  return (f = r)
```



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

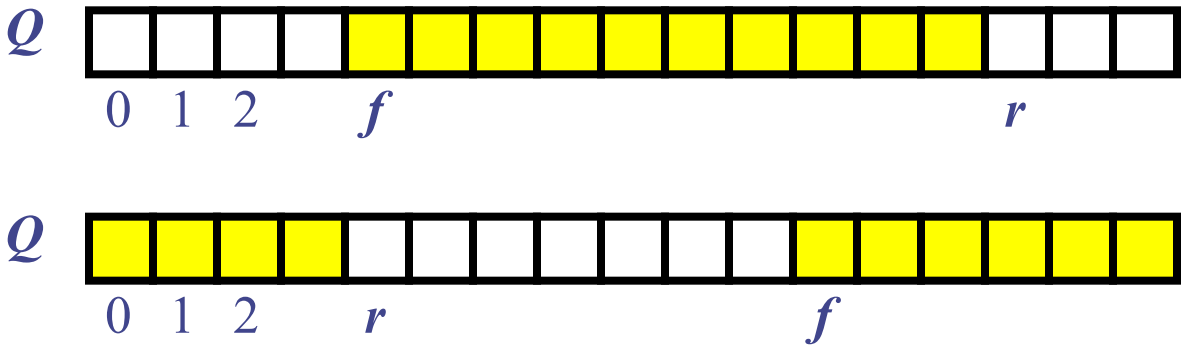
```
Algorithm enqueue(o)
  if size() = N - 1 then
    throw FullQueueException
   else
    Q[r] = o
    r = (r + 1) mod N
```



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
   else
    o = Q[f]
    f = (f + 1) mod N
    return o
```



Performance and Limitations

- array-based implementation of queue ADT

- Performance
  - Let n be the number of elements in the queue
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the queue must be defined *a priori* , and cannot be changed
  - Trying to enqueue an element into a full queue causes an implementation-specific exception

Growable Array-based Queue

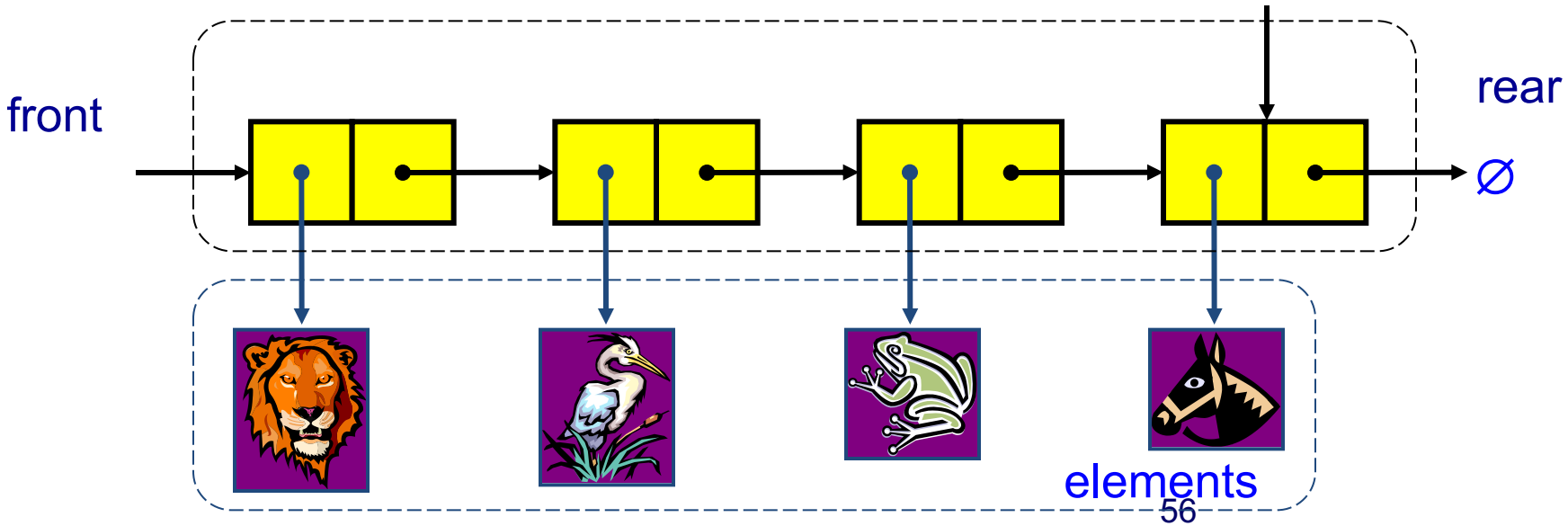
- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- The enqueue operation has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

Exercise

- Describe how to implement a queue using a singly-linked list
  - Queue operations: enqueue(x), dequeue(), size(), isEmpty()
  - For each operation, give the running time

Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the head of the list
  - The rear element is stored at the tail of the list
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time
- NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, I.e., the queue is NEVER full.



Informal C++ Queue Interface

- Informal C++ interface for our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in STL class

```
template <typename Object>
class Queue {
public:
   int size()
   bool isEmpty();
   Object& front()
         throw(EmptyQueueException)
   void enqueue(Object o)
   Object dequeue()
         throw(EmptyQueueException)
};
```

Queue Summary

- Queue Operation Complexity for Different

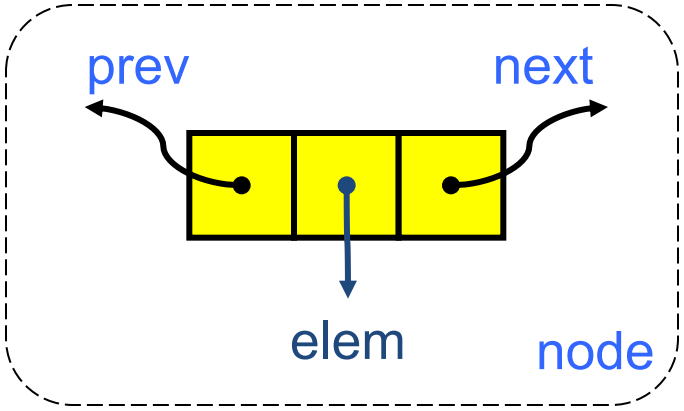
	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
dequeue()	O(1)	O(1)	O(1)
enqueue(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
front()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)

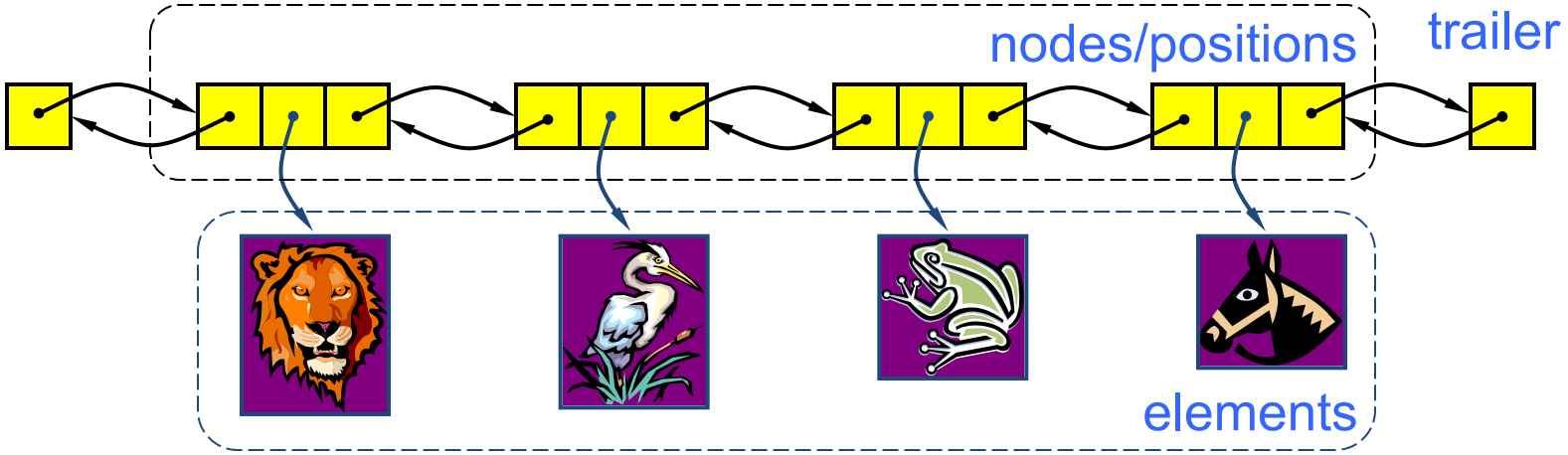
The Double-Ended Queue ADT (§5.3)

- The Double-Ended Queue, or Deque, ADT stores arbitrary objects. (Pronounced 'deck')
- Richer than stack or queue ADTs. Supports insertions and deletions at both the front and the end.
- Main deque operations:
  - insertFirst(object o): inserts element o at the beginning of the deque
  - insertLast(object o): inserts element o at the end of the deque
  - RemoveFirst(): removes and returns the element at the front of the queue
  - RemoveLast(): removes and returns the element at the end of the queue
- Auxiliary queue operations:
  - first(): returns the element at the front without removing it
  - last(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyDequeException

Doubly Linked List

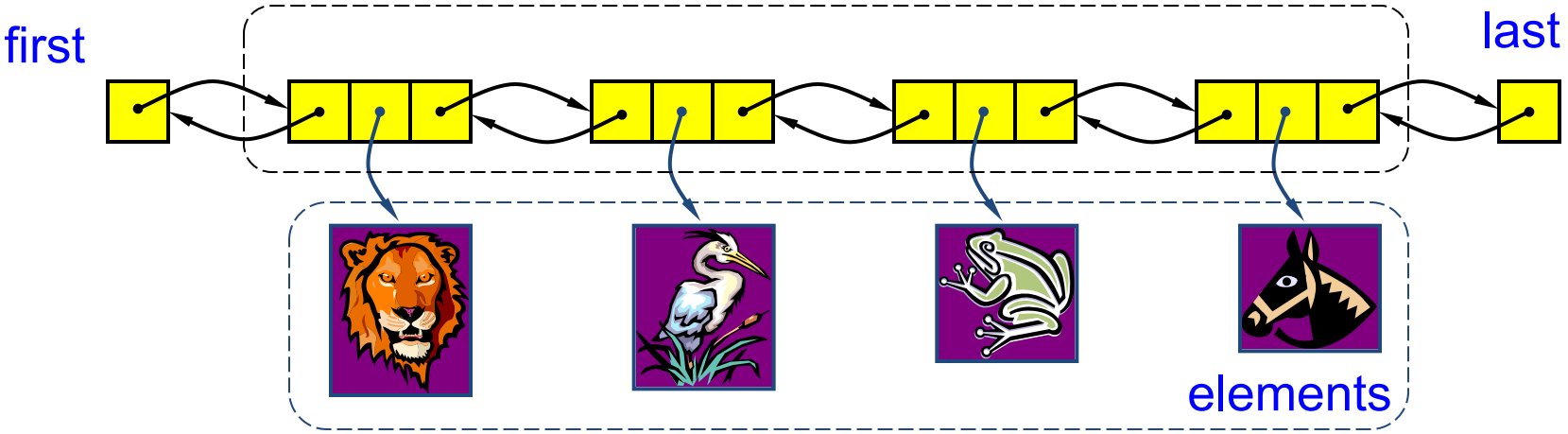
- A doubly linked list provides a natural implementation of the Deque ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





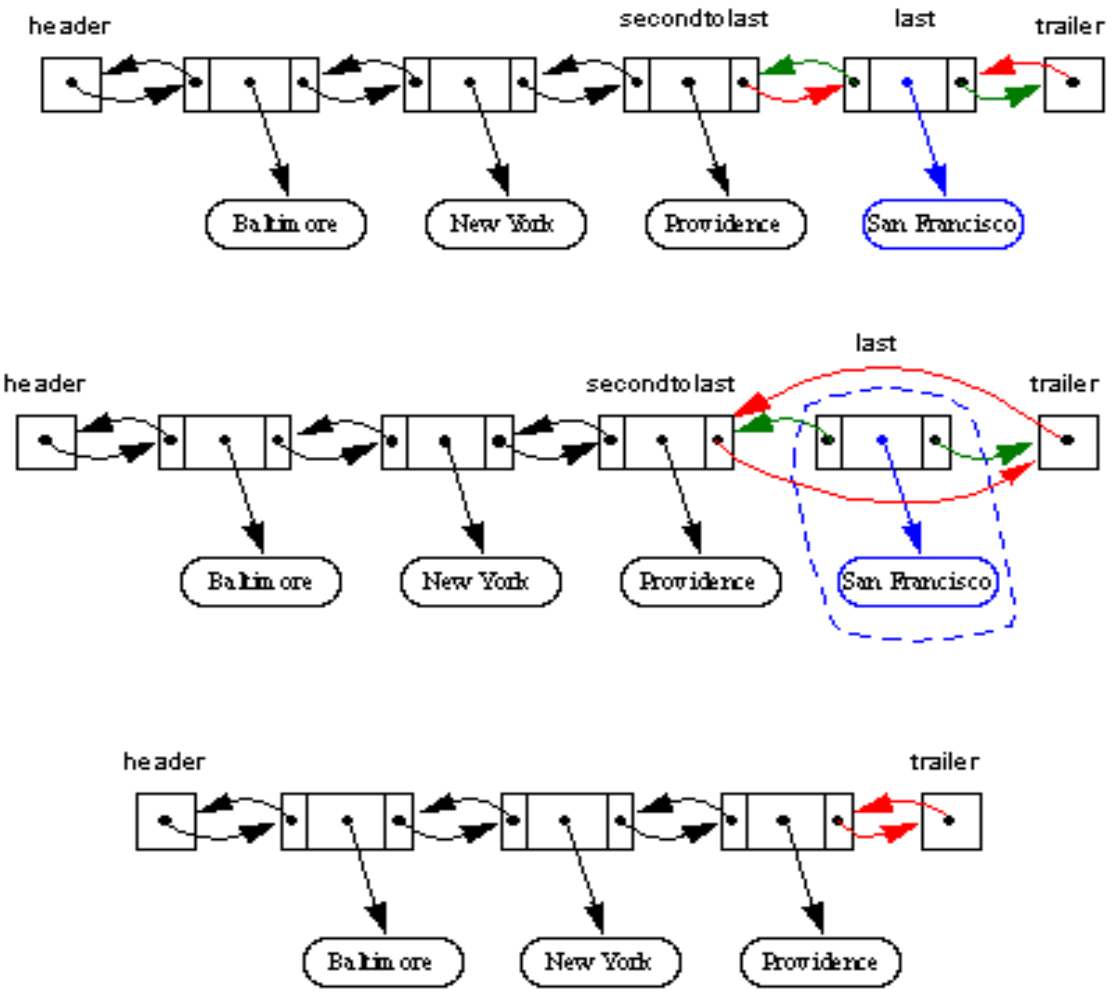
Deque with a Doubly Linked List

- We can implement a deque with a doubly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Deque ADT takes $O(1)$ time



Implementing Deques with Doubly Linked Lists

Here's a visualization of the code for removeLast().



Performance and Limitations

- doubly linked list implementation of deque ADT

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, I.e., the deque is NEVER full.

Deque Summary

- Deque Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked	List Doubly-Linked
removeFirst(), removeLast()	O(1)	O(1)	O(n) for one at list tail, O(1) for other	O(1)
insertFirst(o), InsertLast(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)	O(1)
first(), last	O(1)	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)	O(1)

Implementing Stacks and Queues with Deques

Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()