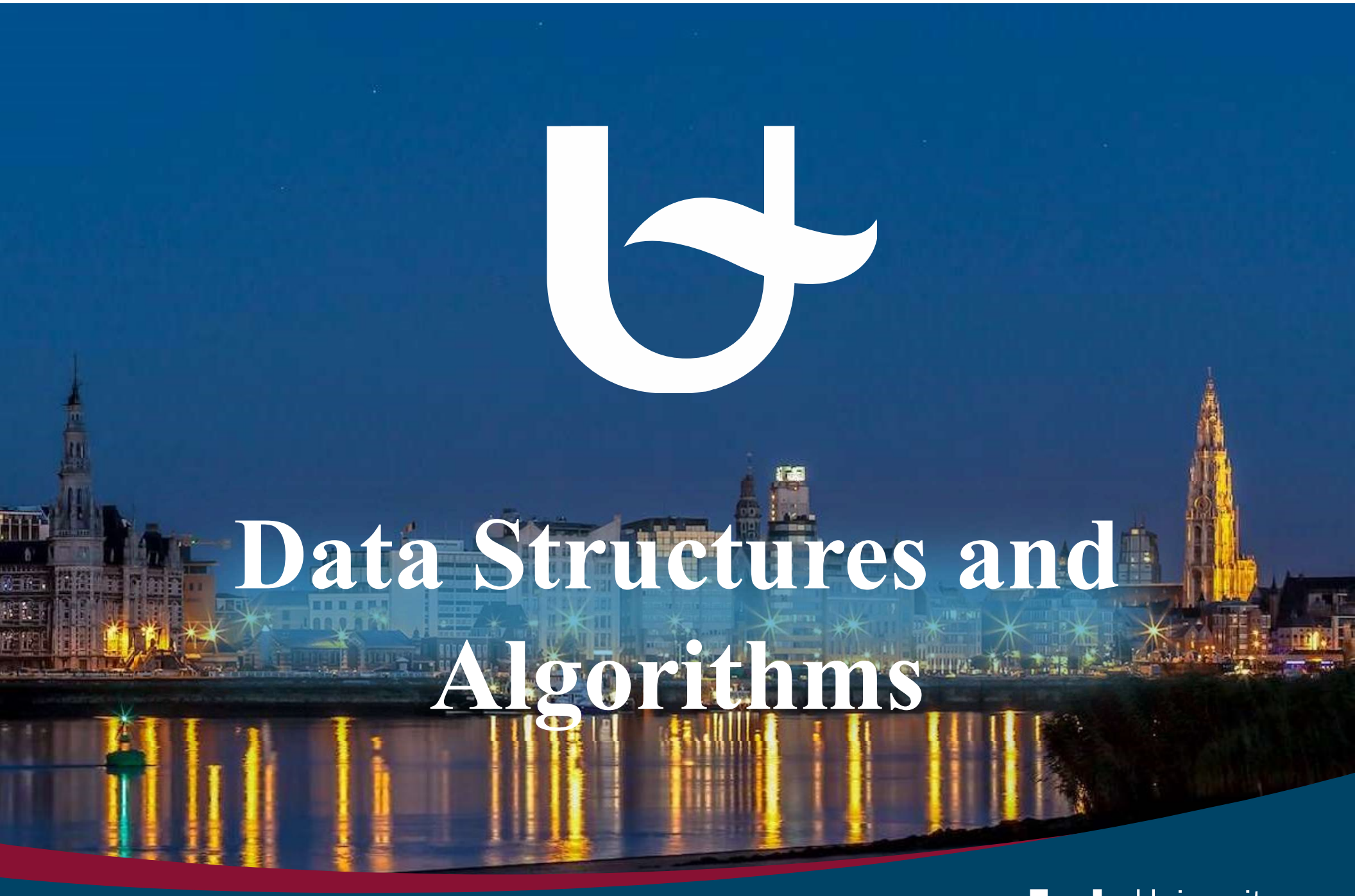


Data Structures and Algorithms

University of Antwerp

List-Based Collections

Lecturer: Dr. Ali Anwar

ADT

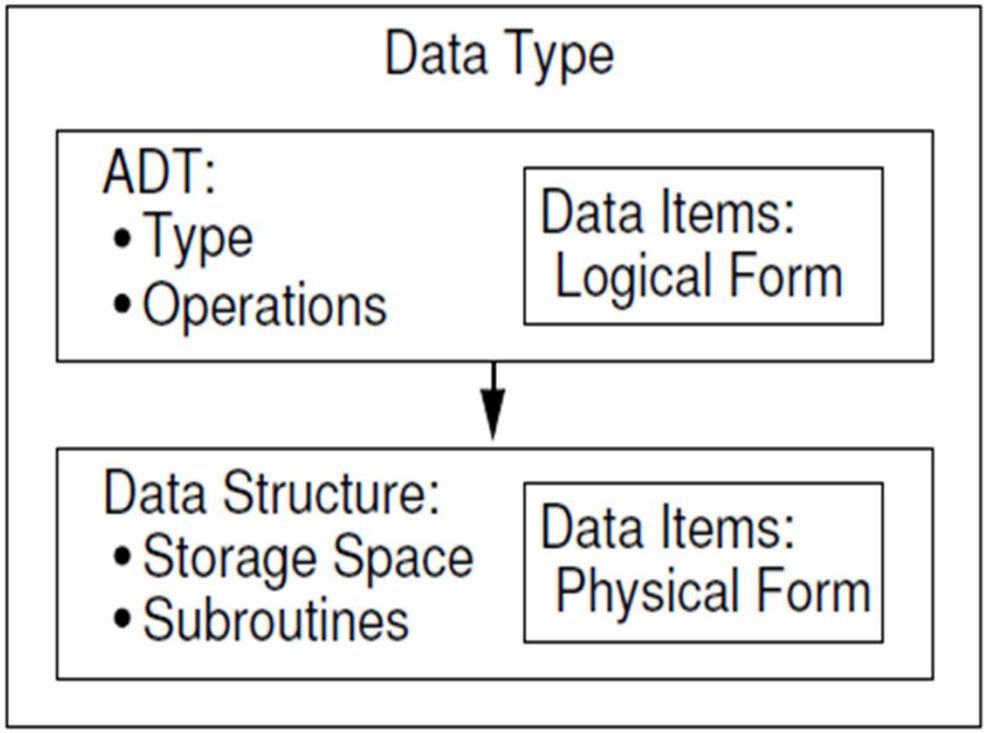


Figure 1.1 The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Objectives

- Design & implementation of
  - Array lists
  - Lists
  - Linked lists
  - Stacks
  - Queues
- Introduction of
  - Iterators
  - Sequences

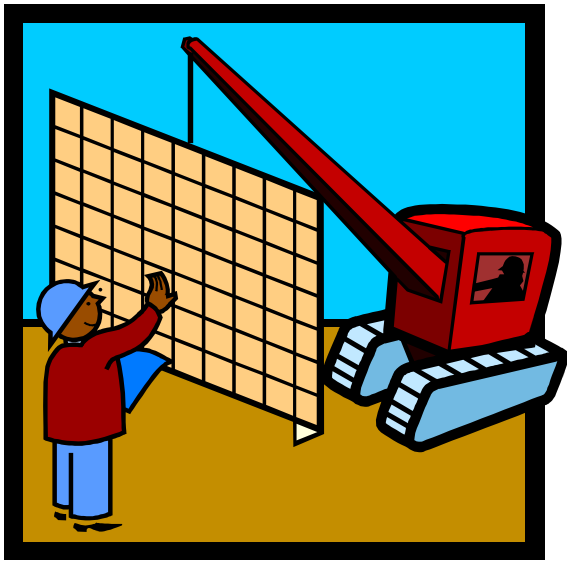
Lecture 2: Learning Outcomes

- What is an array ADT?
- What is amortized cost of the sequence of operations?
- How to calculate the cost of growable array lists?
- What is a Linked List?
- What is a Stack?
- What is a Queue?
- Why do we need these things?

Part 1

Arrays

Goodrich Sec 3.1



The Array List ADT

- The Vector or Array List ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)

- Main methods:
  - *at*(integer *i*): returns the element at index *i* without removing it
  - *set*(integer *i*, object *o*): replace the element at index *i* with *o*
  - *insert*(integer *i*, object *o*): insert a new element *o* to have index *i*
  - *erase*(integer *i*): removes element at index *i*
- Additional methods:
  - *size*()
  - *empty*()

Applications of Array Lists

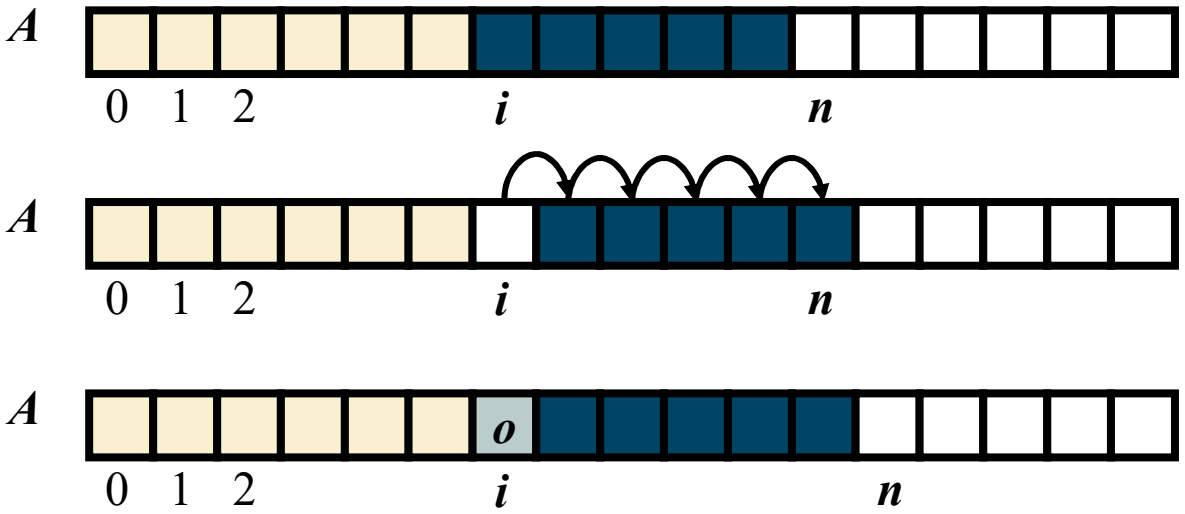
- **Direct applications**
  - Sorted collection of objects (elementary database)
- **Indirect applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Implementation

- Use an array *A* of size *N*
- A variable *n* keeps track of the size of the array list (number of elements stored)
- Operation *at(i)* is implemented in *O(1)* time by returning *A[i]*
- Operation *set(i,o)* is implemented in *O(1)* time by performing *A[i] = o*

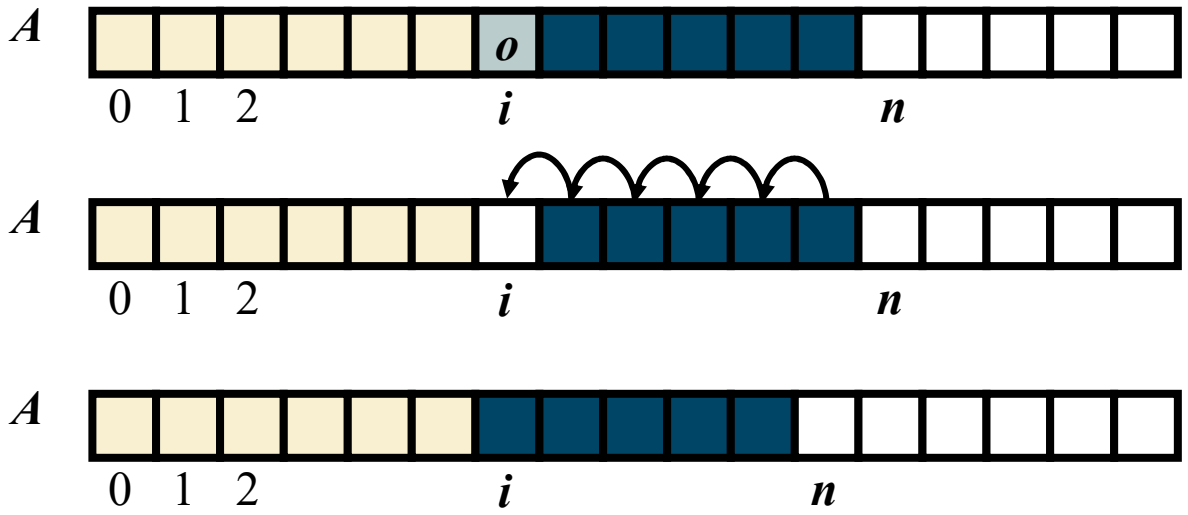
Insertion

- In operation *insert(i, o)*, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n - 1]$
- In the worst case $(i = 0)$, this takes $O(n)$ time



Element Removal

- In operation *erase*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In the array based implementation of an array list:
  - The space used by the data structure is O(n)
  - *size*, *empty*, *at* and *set* run in O(1) time
  - *insert* and *erase* run in O(n) time in worst case
- If we use the array in a circular fashion, operations *insert(0, x)* and *erase(0, x)* run in O(1) time *(will be explained in Queues)*
- In an *insert* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Growable Array-based Array List

- In an *insert*(*o*) operation (without an index), we always insert at the end
- When the array is full, we 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 ***insert*(*o*)**
  if ***t* = *S.length* – 1 then**
    ***A* ← new array of size … (n+1 | 2n)**
    for ***i*** ← 0 **to** ***n*-1 do**
      ***A*[*i*] ← *S*[*i*]**
    ***S* ← *A* (point *A* to *S*)**
  ***n* ← *n* + 1**
  S*[*n*-1] ← *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 *insert*(o) operations
- We assume that we start with an empty *stack* represented by an array of size 1
- We call amortized time of an *insert* operation the average time taken by an insert over the series of operations, i.e., $\frac{T(n)}{n}$

Incremental Strategy Analysis

- We replace the array $k = \frac{n}{c}$ times for $n > c \geq 1$
- The total time $T(n)$ of a series of n insert 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 an *insert* operation is $O(n)$

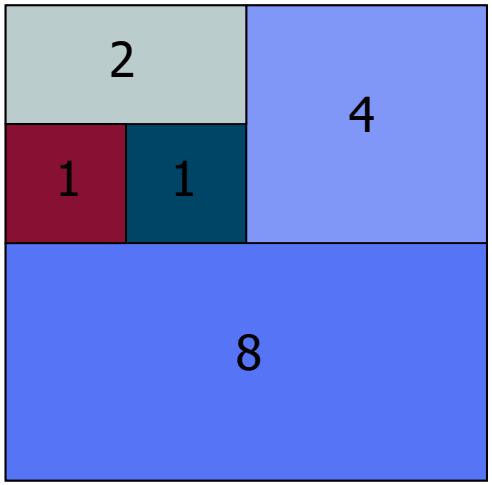
Doubling Strategy Analysis

- We replace the array $k = log_2 n$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 = 3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an *insert* operation is $O(1)$

geometric series



Part 2

Linked Lists

Goodrich Sec 3.2, 3.3

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()*

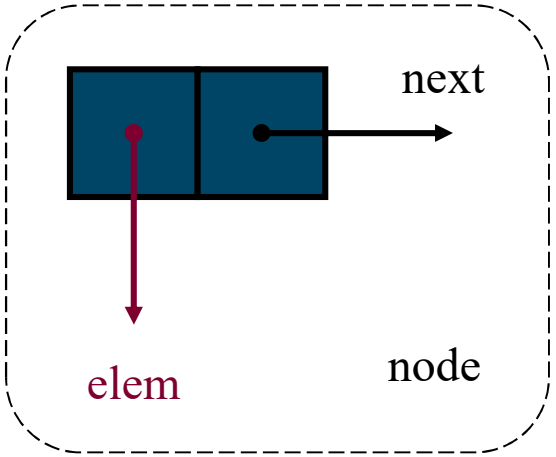
Iterator-based update:

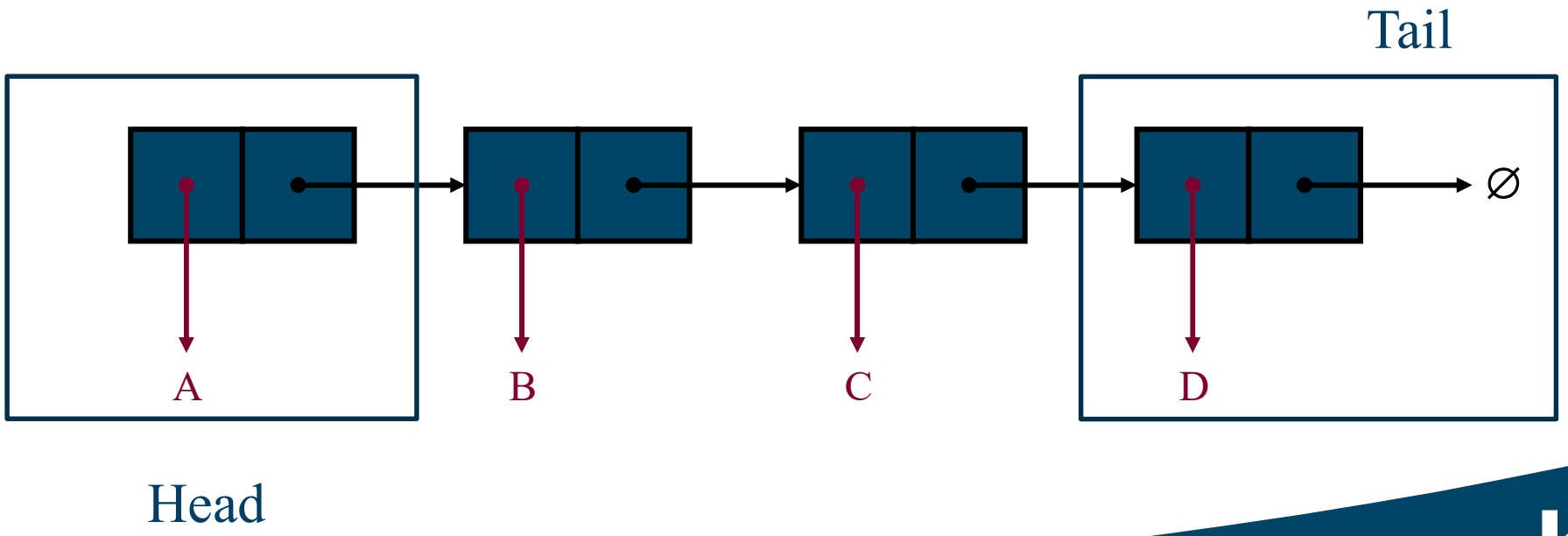
- *insert(p, e)*
- *remove(p)*

* Node list includes the positional ADT

Singly Linked List

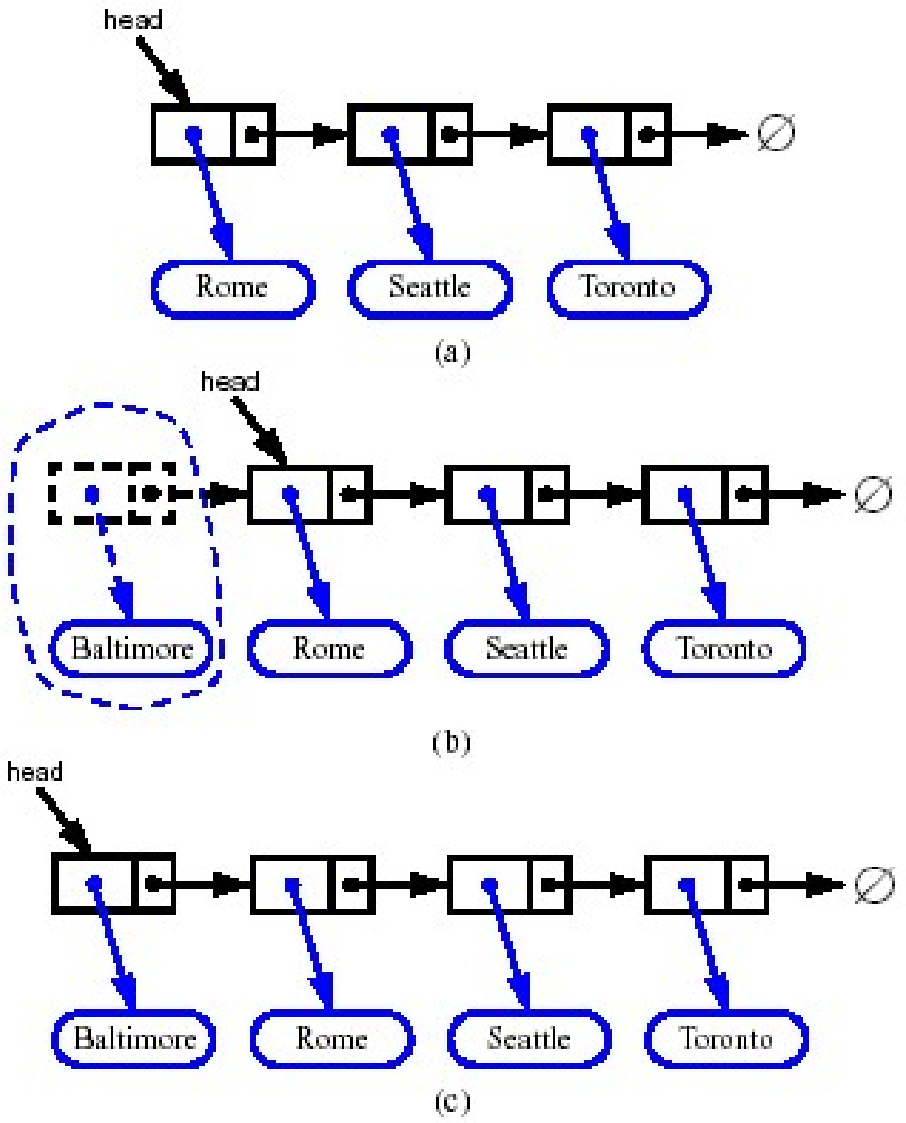
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





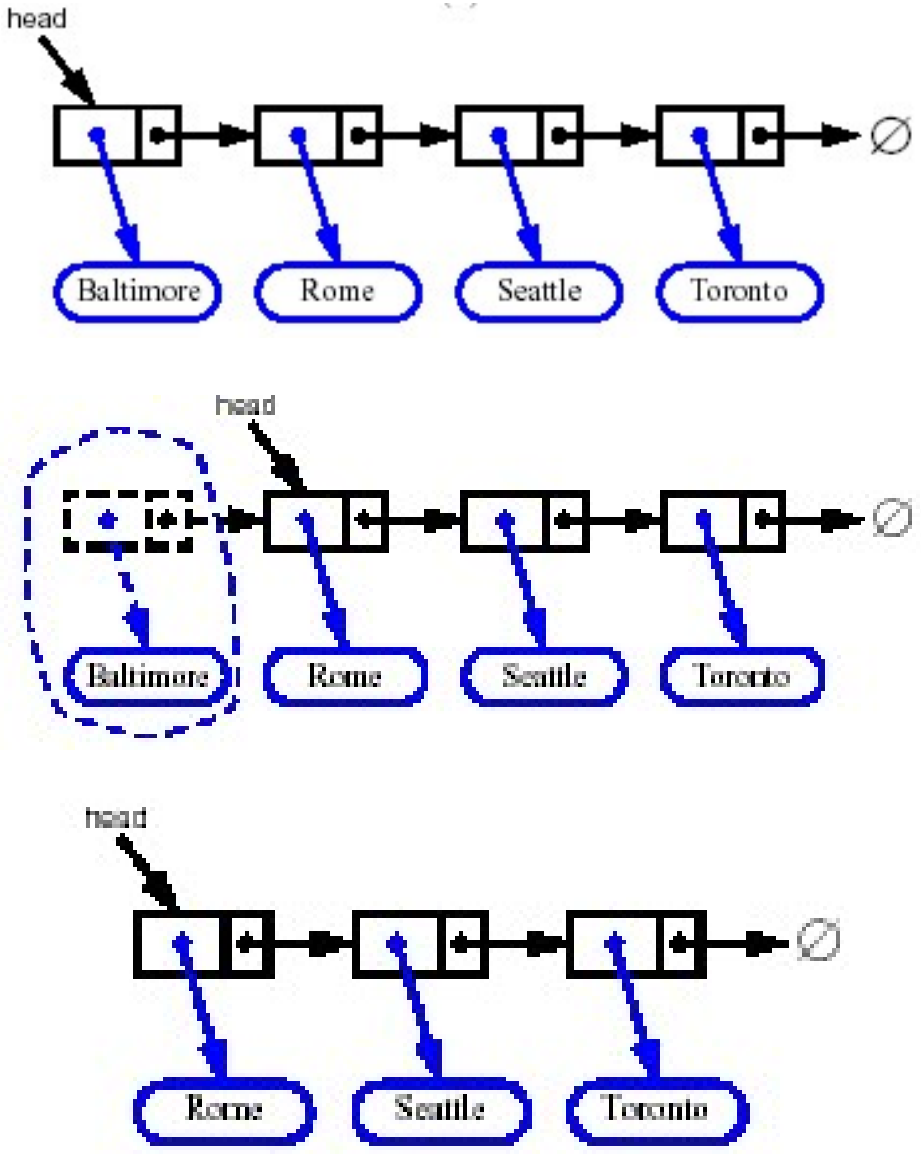
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



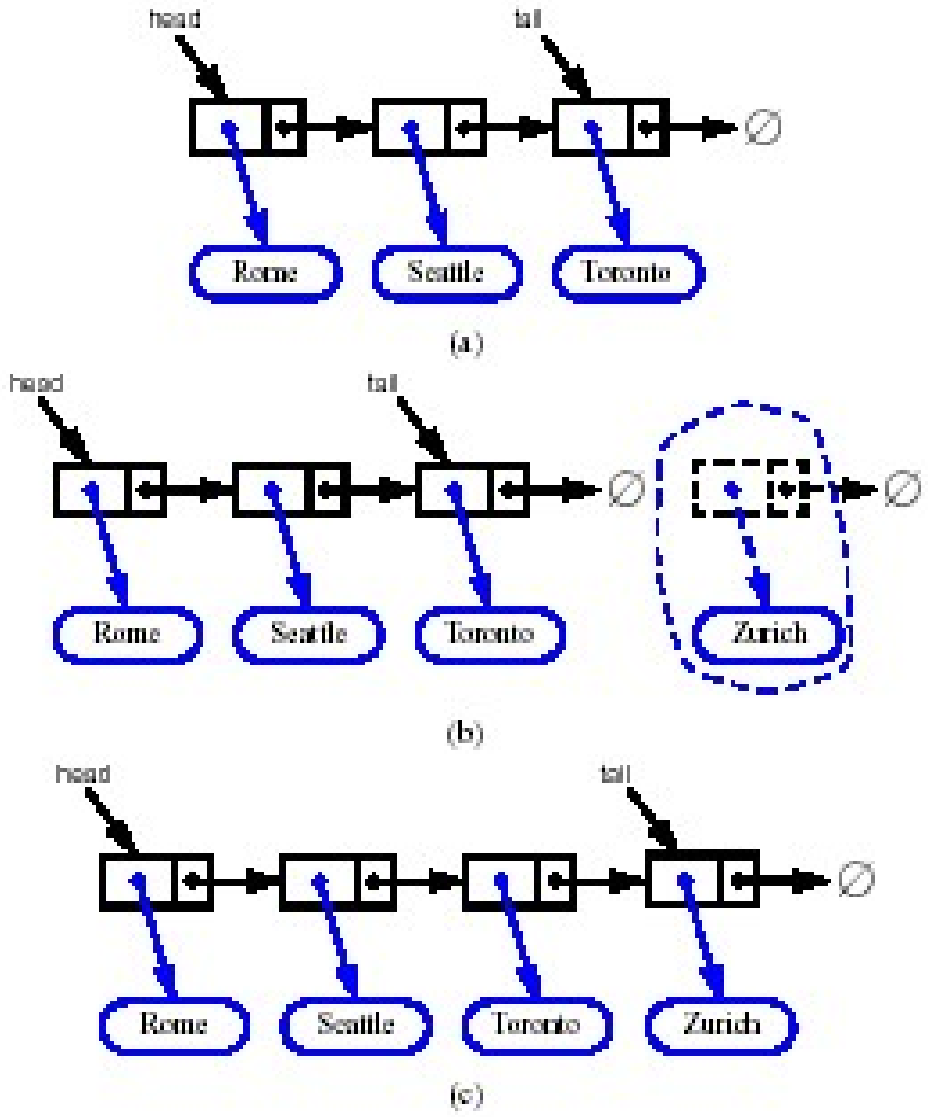
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



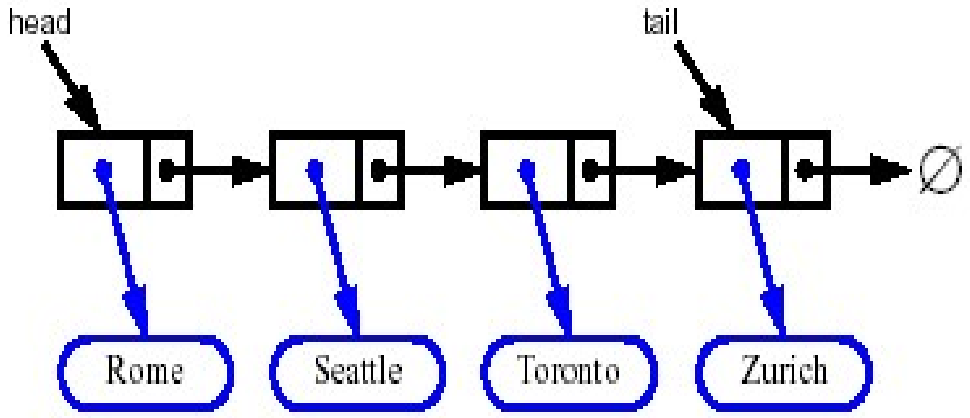
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node
- *You have to traverse the whole Linked List to find the end / tail*



Insertion and removal in general

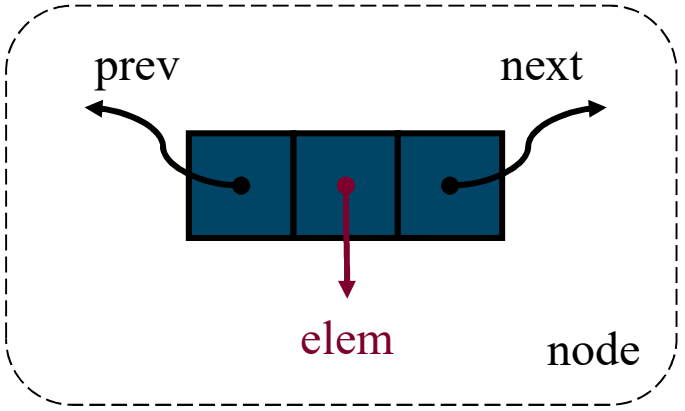
- Do the steps:
  - Traverse the Linked List
  - Fix the links
  - For removal: Free the memory (for C / C++)

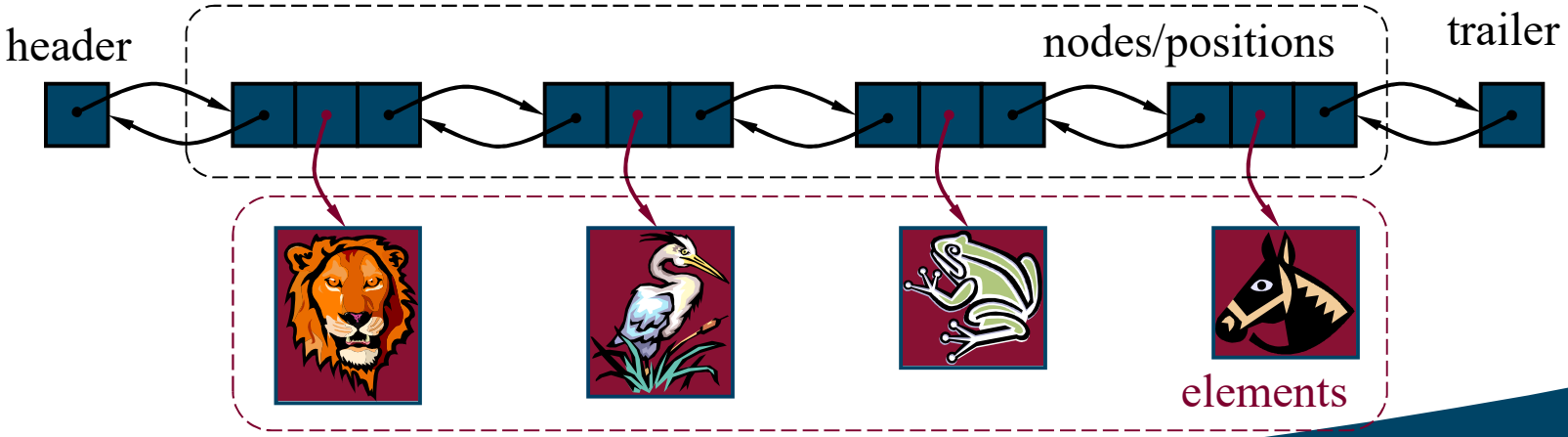
Costs of Singly Linked List

- Traversal O(n)
- Insertion O(n)
- Deletion O(n)

Doubly Linked List

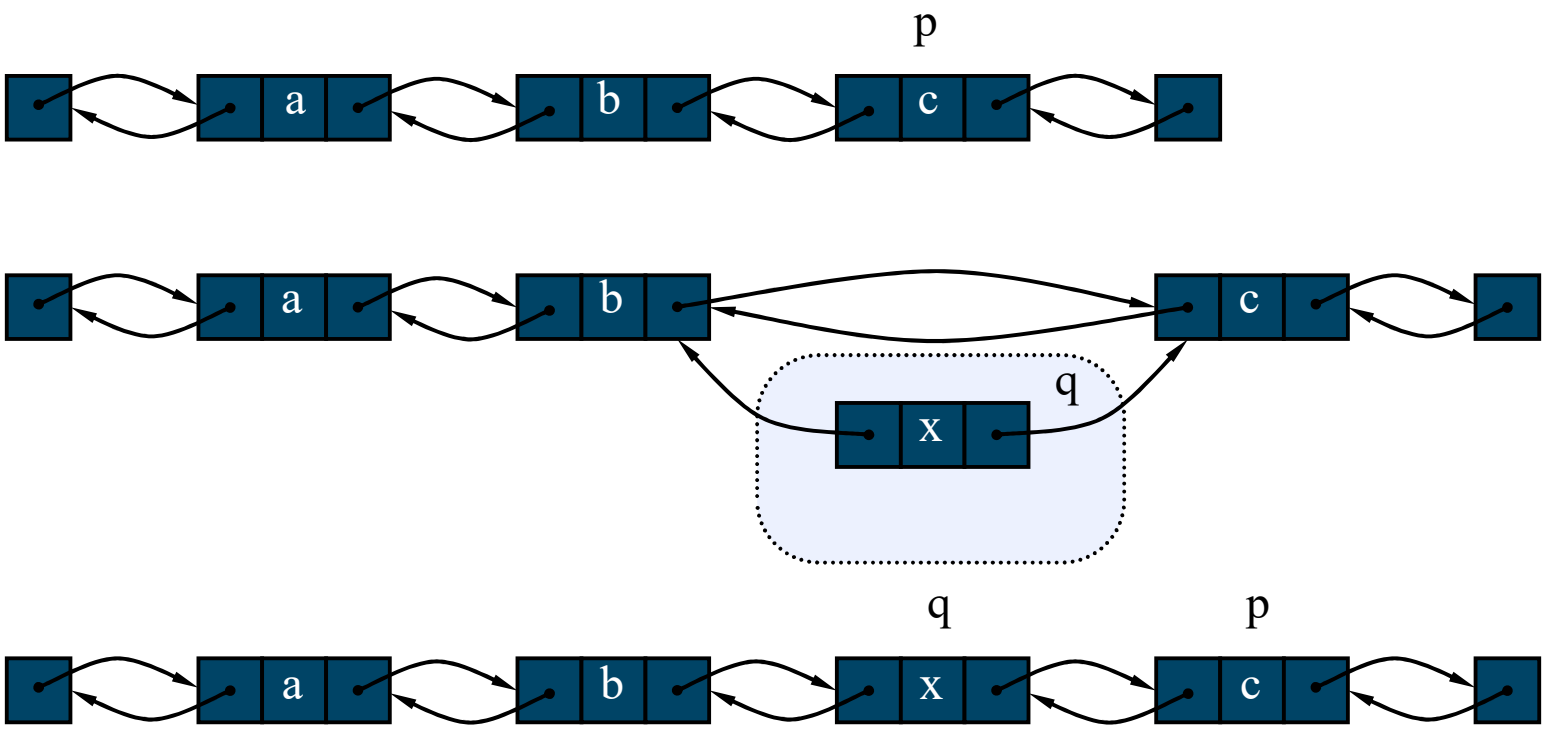
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
- Ability to do a reverse lookup





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

$v \rightarrow$ element = e

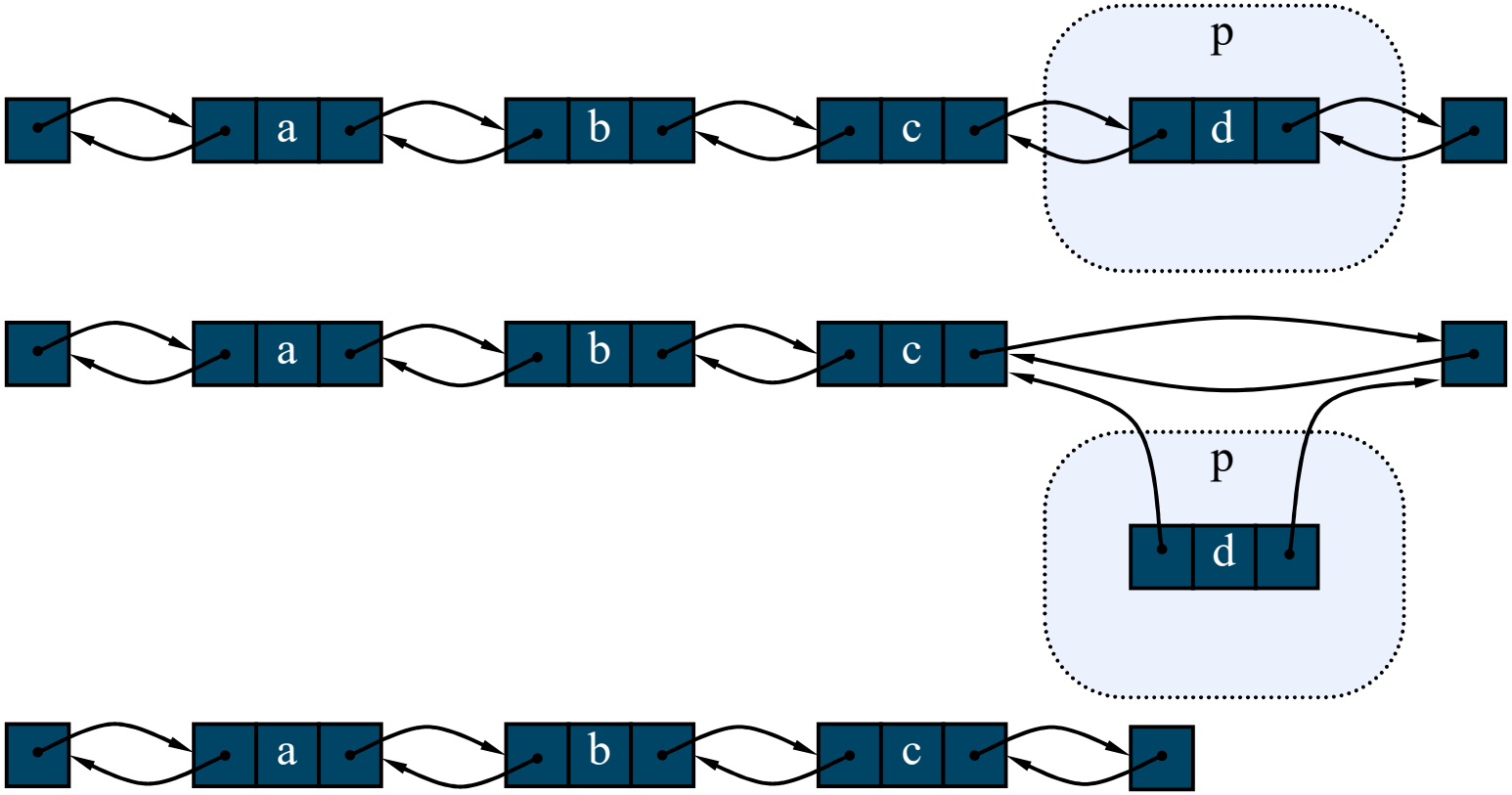
$u = p \rightarrow$ prev

$v \rightarrow$ next = p; $p \rightarrow$ prev = v {link in v before p}

$v \rightarrow$ prev = u; $u \rightarrow$ next = v {link in v after u}

Deletion

- We visualize operation *remove*(*p*)



Deletion Algorithm

Algorithm *remove*(p):

u = p → prev

w = p → next

u → next = w {linking out p}

w → prev = u

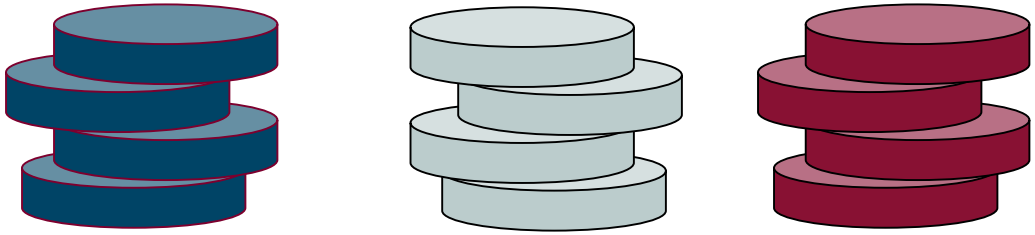
Performance

- 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 $O(1)$
  - Insertion and deletion operations of the List ADT run in $O(1)$ time
    - Traversal still runs in O(n) time
  - Operation *element*() of the Position ADT runs in $O(1)$ time

Part 3

Stacks

Goodrich Sec 5.1



ADT (Abstract Data Types)

- 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
- Example: ADT modeling a simple stock trading system
  - The data stored are *buy*/*sell* orders
  - The operations supported are
    - order *buy*(stock, shares, price)
    - order *sell*(stock, shares, price)
    - void *cancel*(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

Stack ADT

- The **Stack** ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - *push*(object): inserts an element
  - object *pop*(): removes the last inserted element
- Auxiliary stack operations:
  - object *top*(): returns the last inserted element without removing it
  - integer *size*(): returns the number of elements stored
  - boolean *empty*(): indicates whether no elements are stored

Stack Interface in C++

- C++ interface corresponding to our Stack ADT
- Uses an exception class StackEmpty
- Different from the built-in C++ STL class stack

```
template <typename E>
class Stack  {
public:
   int size() const;
   bool empty() const;
   const E& top() const
      throw(StackEmpty);
   void push(const E& e);
   void pop()
      throw(StackEmpty);
}
```

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 pop or top on an empty stack throws a StackEmpty exception

Application of Stacks

- **Direct applications**
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the C++ run-time system
- **Indirect applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stacks

- 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 StackEmpty
    else
        t ← t – 1
        return S[t + 1]
```

Array-based Stacks (cont.)

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

Algorithm *push*(*o*)
  if *t* = *S.size*() – 1 **then**
    throw *StackFull*
  else
    t ← *t* + 1
    S[*t*] ← *o*

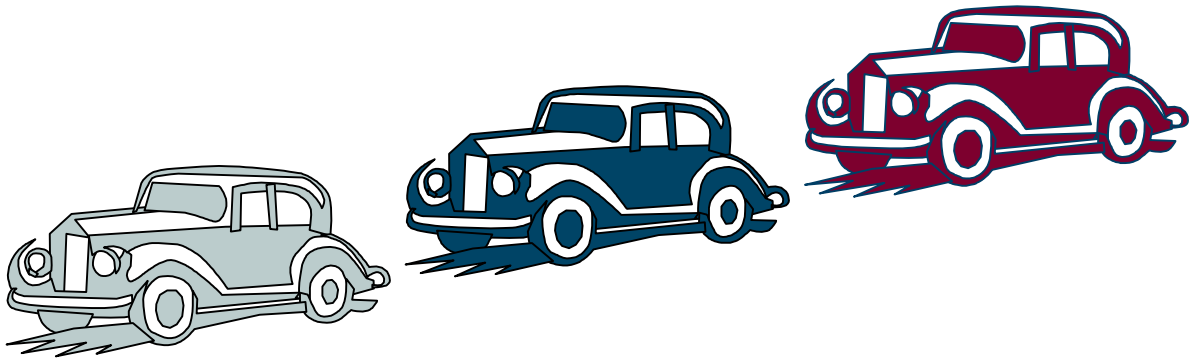
Performance and Limitations

- 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 → in case of array-based implementation
  - Trying to push a new element into a full stack causes an implementation-specific exception

Part 4

Queues

Goodrich Sec 5.2



Queue ADT

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

Example

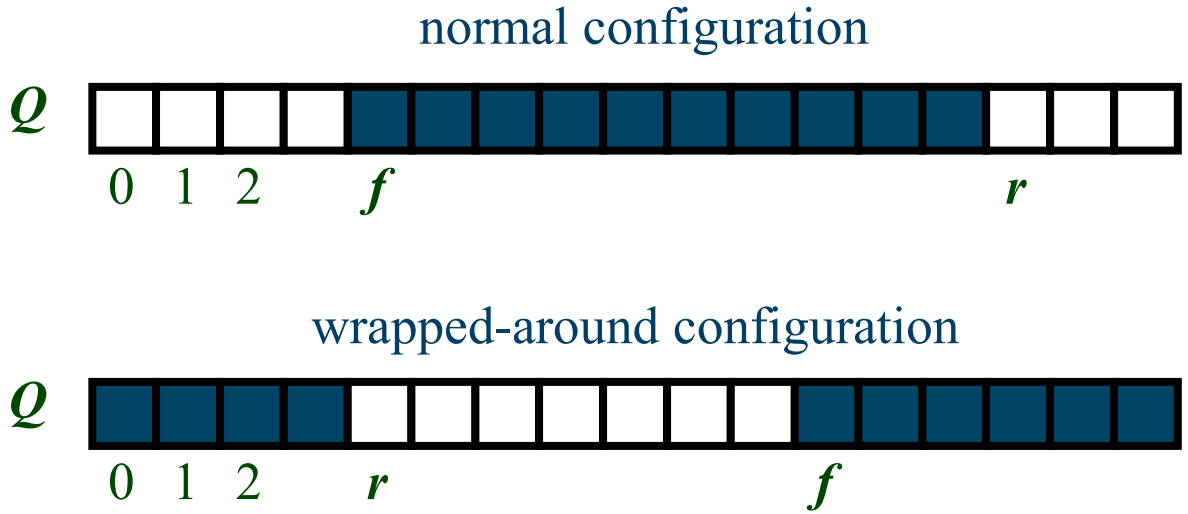
Operation	***Output***	***Q***
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	–	(3)
enqueue(7)	–	(3, 7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	–	(7, 3, 5)

Application of Queues

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

Array-based Queues

- Use an array of size N in a circular fashion
- Three variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
  - n number of items in the queue

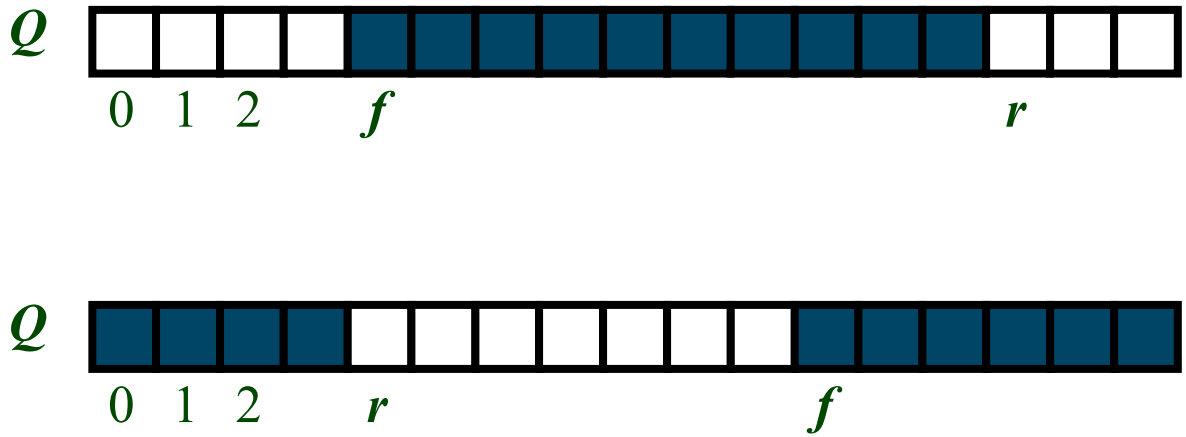


Queue Operations

- Use n to determine size and emptiness

Algorithm ***size***()
    return ***n***

Algorithm ***empty***()
    return (***n*** = **0**)

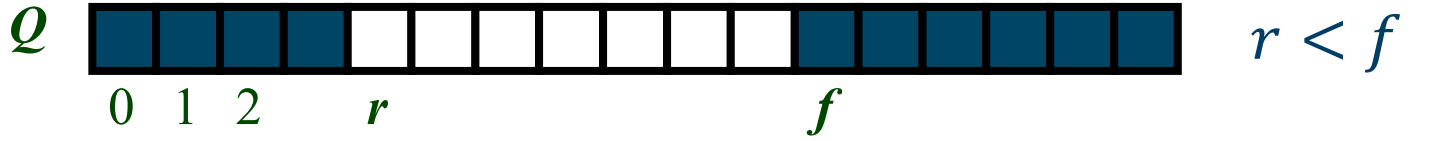


Queue Operations (cont.)

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

Algorithm ***enqueue***(***o***)
    if ***size***() = ***N*** – 1 **then**
        throw **QueueFull**
    else
        Q[***r***] ← ***o***
        r ← (***r*** + 1) mod ***N***
        n ← ***n*** + 1

$f \leq r$

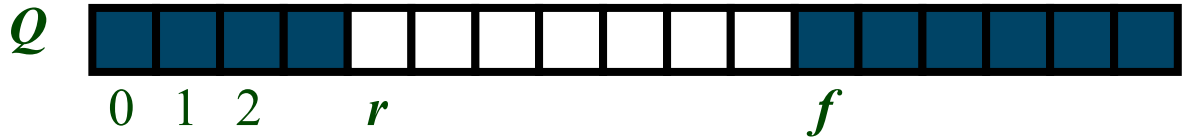


$r < f$

Queue Operations (cont.)

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

```
Algorithm dequeue(o)
    if empty() then
        throw QueueEmpty
    else
        f ← (f + 1) mod N
        n ← n – 1
```



Queue Interface in C++

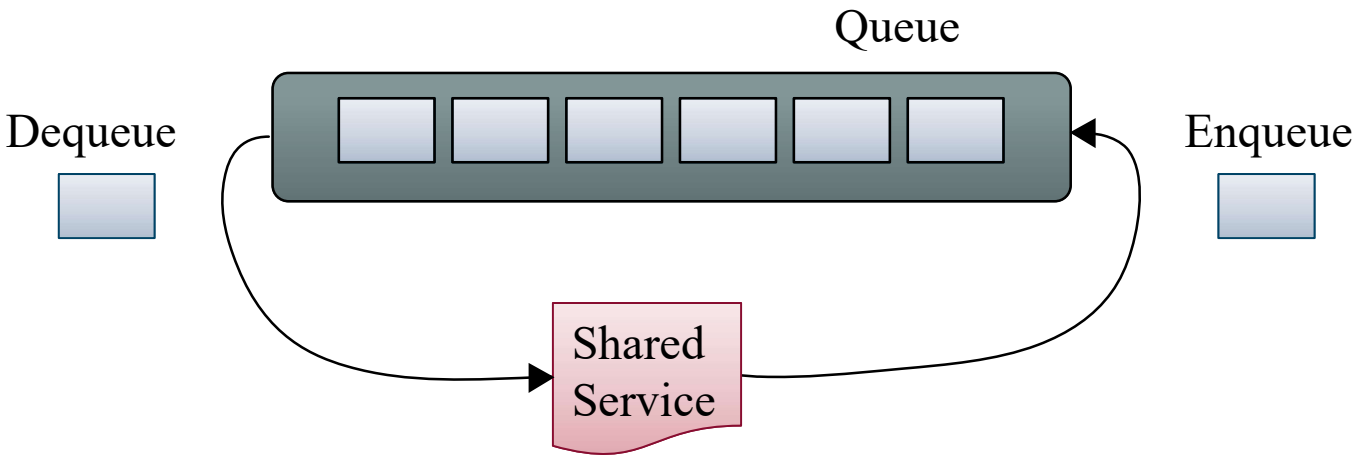
- C++ interface corresponding to our Queue ADT
- Requires the definition of exception QueueEmpty
- No corresponding built-in C++ class

```
template <typename E>
class Queue {
public:
  int size() const;
  bool empty() const;
  const E& front() const
    throw(QueueEmpty);
  void enqueue (const E& e);
  void dequeue()
    throw(QueueEmpty);
};
```


Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:

1. $e = Q.front(); Q.dequeue()$
2. Service element e
3. $Q.enqueue(e)$



Part 5

Containers and 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*)

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*()
- Array List-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*)

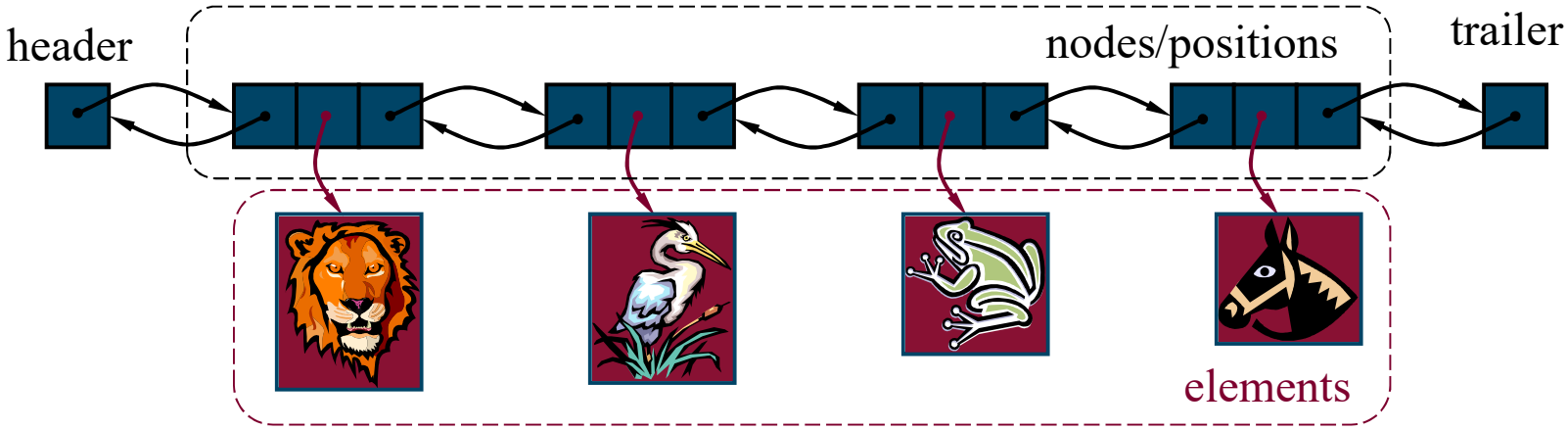
Application of Sequences

- 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

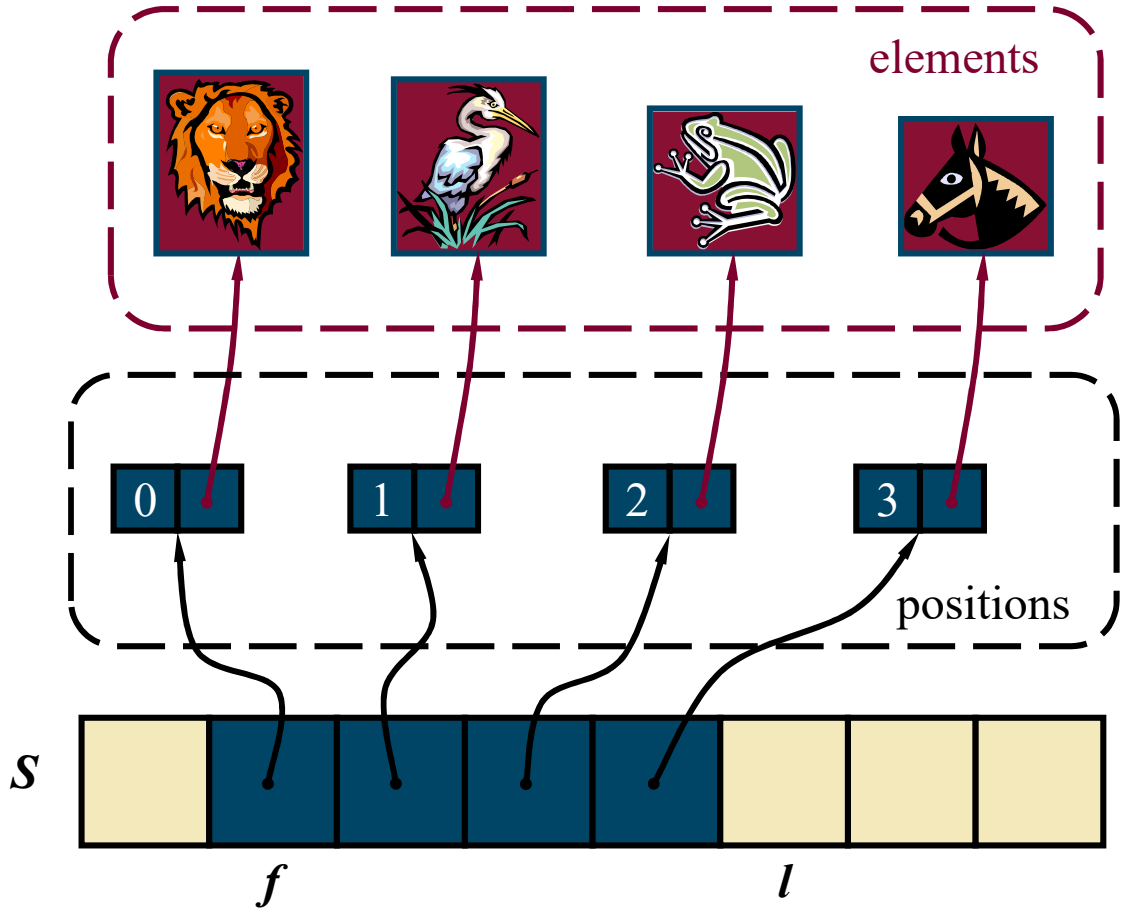
- A doubly linked list provides a reasonable implementation of the Sequence ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

- 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 O(min(i+1, n-i))



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

Operation	Array	List
size, empty	1	1
atIndex, indexOf, at	1	***n***
begin, end	1	1
set(p,e)	1	1
set(i,e)	1	***n***
insert(i,e), erase(i)	***n***	***n***
insertBack, eraseBack	1	1
insertFront, eraseFront	***n***	1
insert(p,e), erase(p)	***n***	1