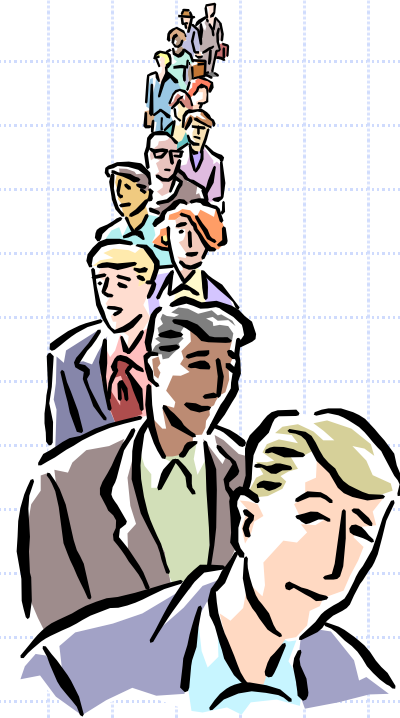


Iterators and Sequences



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

- ❑ The conventional way to iterate through an STL vector

```
int vectorSum1(const vector<int>& V) {  
    int sum = 0;  
    for (int i = 0; i < V.size(); i++)  
        sum += V[i];  
    return sum;  
}
```

Code Fragment 6.13: A simple C++ function that sums the entries of an STL vector.

Iterating through a Container

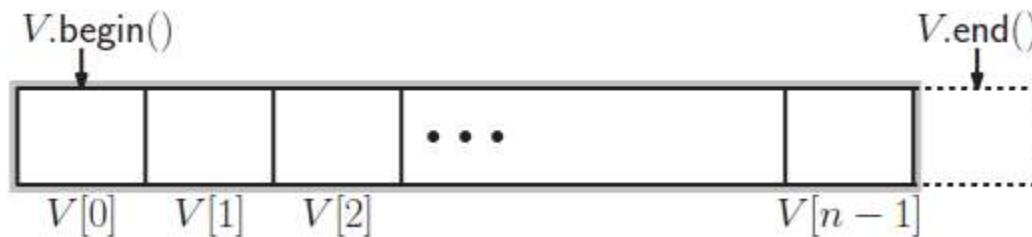
- Let A be a container and p be an iterator for C
for (p = A.begin(); p != A.end(); ++p)
 loop_body
- Use of an iterator to compute the sum of an STL vector

```
int vectorSum2(vector<int> V) {  
    typedef vector<int>::iterator Iterator;           // iterator type  
    int sum = 0;  
    for (Iterator p = V.begin(); p != V.end(); ++p)  
        sum += *p;  
    return sum;  
}
```

Code Fragment 6.14: Using an iterator to sum the elements of an STL vector.

Implementing Iterators

- Array-based
 - Array A of n elements
 - Index i that keeps track of the cursor
 - $\text{begin}() = 0$
 - $\text{end}() = n$ (index following the last element)



Implementing Iterators

- **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 A supports iterators:
 - `A::iterator` – read/write iterator type
 - `A::const_iterator` – read-only iterator type
 - `A.begin()`, `A.end()` – return start/end iterators
- This iterator-based operators and methods:
 - `*p`: access current element
 - `++p`, `--p`: advance to next/previous element
 - `A.assign(p, q)`: replace A 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)`

STL Vectors and Algorithms

❑ `#include <algorithm>`

`sort(p, q)`: Sort the elements in the range from p to q in ascending order. It is assumed that less-than operator (“<”) is defined for the base type.

`random_shuffle(p, q)`: Rearrange the elements in the range from p to q in random order.

`reverse(p, q)`: Reverse the elements in the range from p to q .

`find(p, q, e)`: Return an iterator to the first element in the range from p to q that is equal to e ; if e is not found, q is returned.

`min_element(p, q)`: Return an iterator to the minimum element in the range from p to q .

`max_element(p, q)`: Return an iterator to the maximum element in the range from p to q .

`for_each(p, q, f)`: Apply the function f the elements in the range from p to q .

Sequence ADT

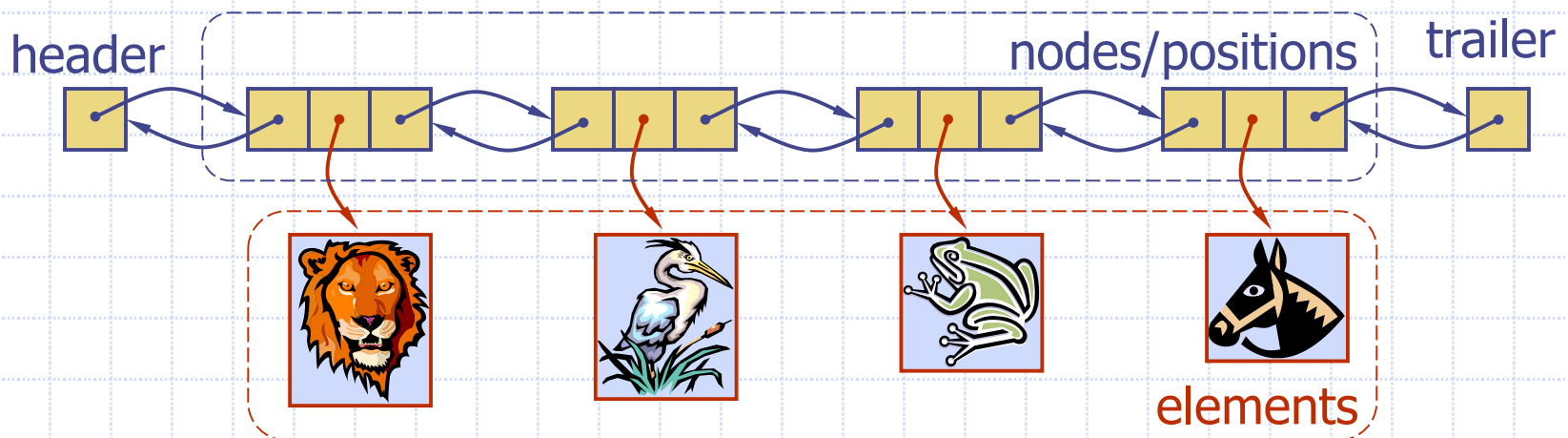
- The **Sequence** ADT is the union of the Array List and Node List ADTs
- Elements accessed by
 - Index, or
 - Position
- Generic methods:
 - **size()**, **empty()**
- ArrayList-based methods:
 - **at(i)**, **set(i, o)**, **insert(i, o)**, **erase(i)**
- List-based methods:
 - **begin()**, **end()**
 - **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

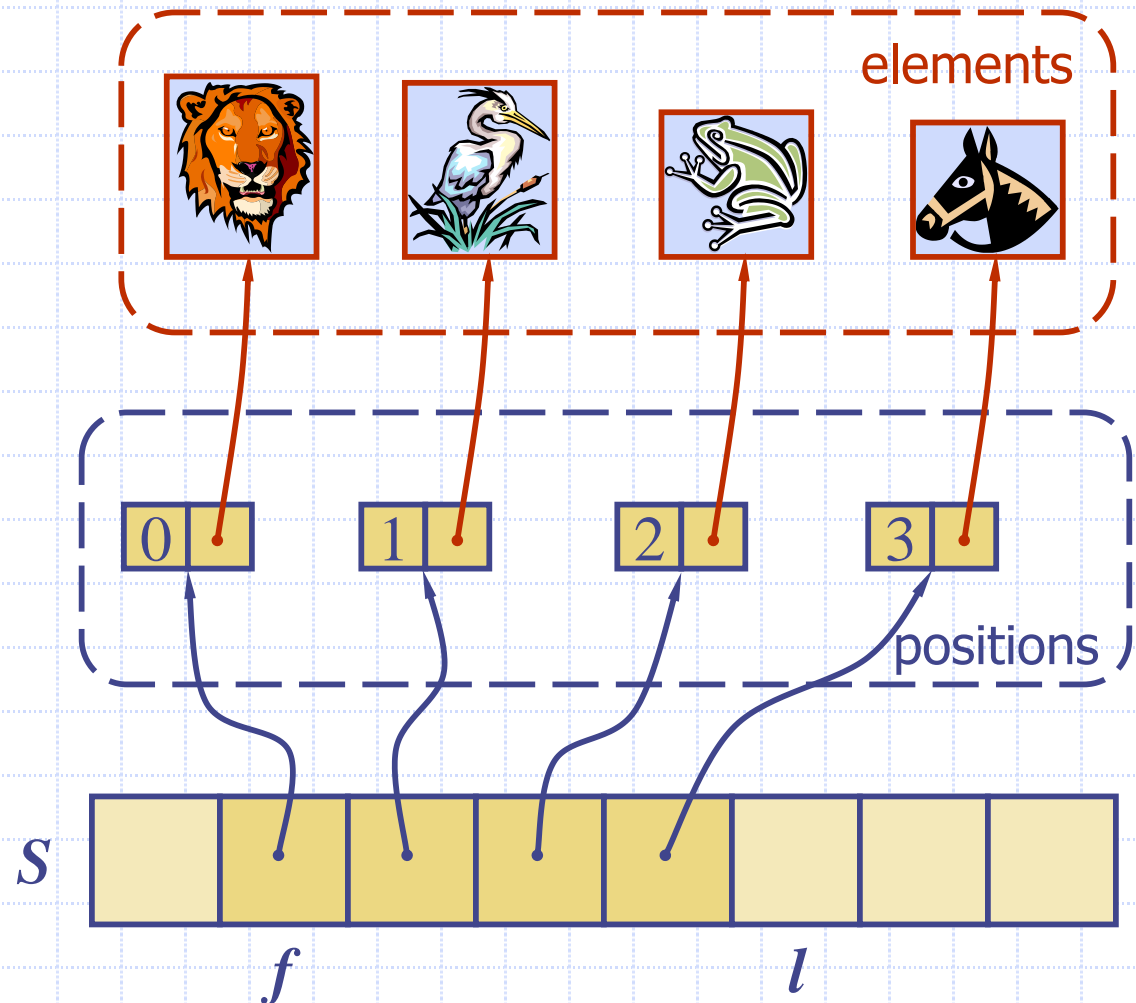
Linked List Implementation

- 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
- 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
- Special trailer and header nodes



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