VECTORS & SEQUENCES

Dr. Megha Ummat

Vectors

- Suppose we have a collection S of n elements stored in a certain linear order, so that we can refer to the elements in S as first, second, third, and so on. Such a collection is generically referred to as a ***list*** or ***sequence***.
- We can uniquely refer to each element e in S using an integer in the range $[0,n-1]$ that is equal to the number of elements of S that precede e in S.
- The ***index*** of an element e in S is the number of elements that are before e in S. Hence, the first element in S has index 0 and the last element has index $n-1$.
- Also, if an element of S has index i, its previous element (if it exists) has index $i-1$, and its next element (if it exists) has index $i+1$.

Vectors

- This concept of index is related to that of the ***rank*** of an element in a list, which is usually defined to be one more than its index; so the first element is at rank 1, the second is at rank 2, and so on.
- A sequence that supports access to its elements by their indices is called a ***vector***.
- Since our index definition is more consistent with the way arrays are indexed in C++ and other common programming languages, we refer to the place where an element is stored in a vector as its "index," rather than its "rank."

Vector Abstract Data Type

- A ***vector***, also called an ***array list***, is an ADT that supports the following fundamental functions (in addition to the standard size() and empty() functions).
- In all cases, the index parameter i is assumed to be in the range $0 \leq i \leq$ size()-1.
- at(i): Return the element of V with index i; an error condition occurs if i is out of range.
- set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

Vector Abstract Data Type

- insert(i,e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.
- erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Operations on Vector

Operation	*Output*	*V*
insert(0,7)	–	(7)
insert(0,4)	–	(4,7)
at(1)	7	(4,7)
insert(2,2)	–	(4,7,2)
at(3)	*"error"*	(4,7,2)
erase(1)	–	(4,2)
insert(1,5)	–	(4,5,2)
insert(1,3)	–	(4,3,5,2)
insert(4,9)	–	(4,3,5,2,9)
at(2)	5	(4,3,5,2,9)
set(3,8)	–	(4,3,5,8,9)

An Array Based Implementation

- An obvious choice for implementing the vector ADT is to use a fixed size array A, where $A[i]$ stores the element at index i.
- We choose the size N of array A to be sufficiently large, and we maintain the number $n < N$ of elements in the vector in a member variable.
- To implement the at(i) operation, for example, we just return $A[i]$.

An Array Based Implementation

Algorithm insert(i, e):
  for $j = n-1, n-2, \ldots, i$ **do**
    $A[j+1] \leftarrow A[j]$ {make room for the new element}
  $A[i] \leftarrow e$
  $n \leftarrow n+1$

Algorithm erase(i):
  for $j = i+1, i+2, \ldots, n-1$ **do**
    $A[j-1] \leftarrow A[j]$ {fill in for the removed element}
  $n \leftarrow n-1$

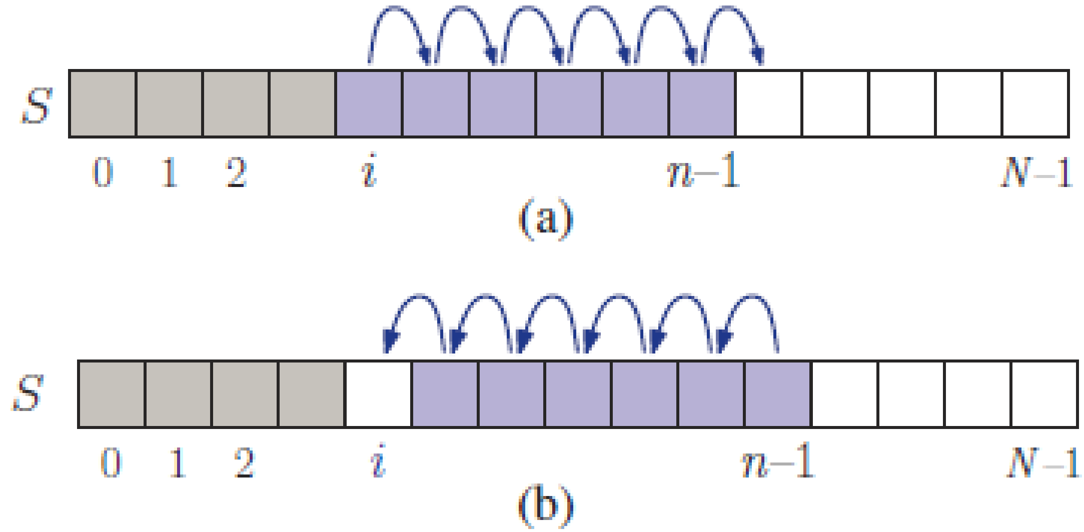


Figure 6.1: Array-based implementation of a vector V that is storing n elements: (a) shifting up for an insertion at index i; (b) shifting down for a removal at index i.

- An important (and time-consuming) part of this implementation involves the shifting of elements up or down to keep the occupied cells in the array contiguous.
- These shifting operations are required to maintain our rule of always storing an element whose list index i at index i in the array A.

Running Times of Array based Implementation

Operation	Time
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i,e)	$O(1)$
insert(i,e)	$O(n)$
erase(i)	$O(n)$

An Extendable Array Implementation

- A major weakness of the simple array implementation for the vector ADT given is that it requires advance specification of a fixed capacity, N, for the total number of elements that may be stored in the vector.
- If the actual number of elements, n, of the vector is much smaller than N, then this implementation will waste space.
- Worse, if n increases past N, then this implementation will crash.

An Extendable Array Implementation

- We provide a means to grow the array A that stores the elements of a vector V. Whenever an ***overflow*** occurs, that is, when n=N and function insert is called, we perform the following steps:
  - Allocate a new array B of capacity $2N$
  - Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, N-1$
  - Deallocate A and reassign A to point to the new array B

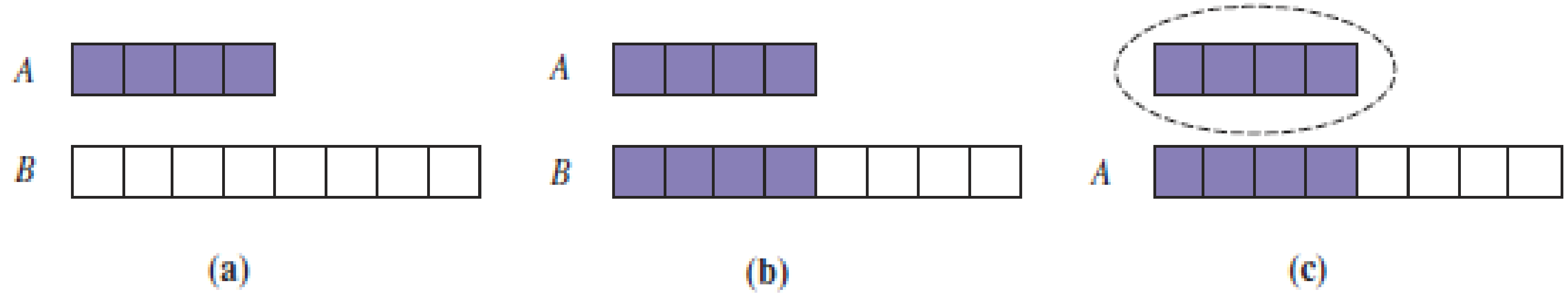


Figure 6.2: The three steps for "growing" an extendable array: (a) create new array B; (b) copy elements from A to B; (c) reassign A to refer to the new array and delete the old array.

An Extendable Array Implementation

- This array replacement strategy is known as an ***extendable array***, for it can be viewed as extending the end of the underlying array to make room for more elements.

```
typedef int Elem;                                    // base element type
class ArrayVector {
public:
  ArrayVector();                                     // constructor
  int size() const;                                  // number of elements
  bool empty() const;                                // is vector empty?
  Elem& operator[](int i);                           // element at index
  Elem& at(int i) throw(IndexOutOfBounds);           // element at index
  void erase(int i);                                 // remove element at index
  void insert(int i, const Elem& e);                 // insert element at index
  void reserve(int N);                               // reserve at least N spots
  // ... (housekeeping functions omitted)
private:
  int capacity;                                      // current array size
  int n;                                             // number of elements in vector
  Elem* A;                                           // array storing the elements
};
```

An Extendable Array Implementation

- The member data for class ArrayVector consists of the array storage *A*, the current number *n* of elements in the vector, and the current storage capacity.
- A new function, called reserve is added, that is not part of the ADT. This function allows the user to explicitly request that the array be expanded to a capacity of a size at least *n*. If the capacity is already larger than this, then the function does nothing.
- Include housekeeping functions (copy constructor, an assignment operator, and a destructor).

An Extendable Array Implementation

- We provide two means for accessing individual elements of the vector.
  - Overriding the C++ array index operator ("[ ]")
  - Using the `at` function.
- The two functions behave the same, except that the `at` function performs a range test before each access. If the index i is not in bounds, this function throws an exception.
- Because both of these access operations return a reference, there is no need to explicitly define a `set` function. Instead, we can simply use the assignment operator.
- For example, the ADT function v.set(i,5) could be implemented either as $v[i] = 5$ or, more safely, as v.at(i) = 5.

- When the vector is constructed, we do not allocate any storage and simply set A to NULL.

- Note that the first attempt to add an element results in array storage being allocated.

```
ArrayVector::ArrayVector()                       // constructor
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const                    // number of elements
  { return n; }

bool ArrayVector::empty() const                  // is vector empty?
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)             // element at index
  { return A[i]; }

                                                 // element at index (safe)
Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}
```

Erase Function

◦ The function erase removes an element at index *i* by shifting all subsequent elements from index *i*+1 to the last element of the array down by one position.

```
void ArrayVector::erase(int i) {            // remove element at index
  for (int j = i+1; j < n; j++)             // shift elements down
    A[j - 1] = A[j];
  n--;                                      // one fewer element
}
```

Reserve function

- The **reserve** function first checks whether the capacity already exceeds n, in which case nothing needs to be done. Otherwise, it allocates a new array B of the desired sizes, copies the contents of A to B, deletes A, and makes B the current array.

```
void ArrayVector::reserve(int N) {          // reserve at least N spots
  if (capacity >= N) return;                // already big enough
  Elem* B = new Elem[N];                    // allocate bigger array
  for (int j = 0; j < n; j++)               // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;               // discard old array
  A = B;                                    // make B the new array
  capacity = N;                             // set new capacity
}
```

Insert Function

- The `insert` function first checks whether there is sufficient capacity for one more element.
- If not, it sets the capacity to the maximum of 1 and twice the current capacity. Then starting at the insertion point, it shifts elements up by one position, and stores the new element in the desired position.

```
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                          // overflow?
    reserve(max(1, 2 * capacity));            // double array size
  for (int j = n − 1; j >= i; j−−)            // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                   // put in empty slot
  n++;                                        // one more element
}
```

Run Time

- In terms of efficiency, this array replacement strategy might, at first, seem rather slow. After all, performing just one array replacement required by an element insertion takes O(n) time, which is not very good.
- Notice, however, that, after we perform an array replacement, our new array allows us to add n new elements to the vector before the array must be replaced again.
- This simple observation allows us to show that the running time of a series of operations performed on an initially empty vector is proportional to the total number of elements added.

Run Time

- As a shorthand notation, let us refer to the insertion of an element meant to be the last element in a vector as a “push” operation.
- Using a design pattern called ***amortization***, we show below that performing a sequence of push operations on a vector implemented with an extendable array is quite efficient.
- **Amortized Analysis** is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

- **Let *V* be a vector implemented by means of an extendable array *A*, as described above. The total time to perform a series of *n* push operations in *V*, starting from *V* being empty and *A* having size *N* = 1, is *O*(*n*).**

- To perform this analysis, we view the computer as a coin-operated appliance, which requires the payment of one ***cyber-dollar*** for a constant amount of computing time.
- When an operation is executed, we should have enough cyber-dollars available in our current "bank account" to pay for that operation's running time.
- Thus, the total amount of cyber-dollars spent for any computation is proportional to the total time spent on that computation.
- The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

- **Let V be a vector implemented by means of an extendable array A, as described above. The total time to perform a series of n push operations in V, starting from V being empty and A having size $N = 1$, is $O(n)$.**
- Let us assume that one cyber-dollar is enough to pay for the execution of each push operation in V, excluding the time spent for growing the array.
- Also, let us assume that growing the array from size k to size $2k$ requires k cyber-dollars for the time spent copying the elements. We shall charge each push operation three cyber-dollars.
- Thus, we overcharge each push operation that does not cause an overflow by two cyber-dollars. Think of the two cyber-dollars profited in an insertion that does not grow the array as being "stored" at the element inserted.
- An overflow occurs when the vector V has 2^i elements, for some $i \geq 0$, and the size of the array used by V is 2^i. Thus, doubling the size of the array requires 2^i cyber-dollars.
- Fortunately, these cyber-dollars can be found at the elements stored in cells 2^{i-1} through $2i - 1$.

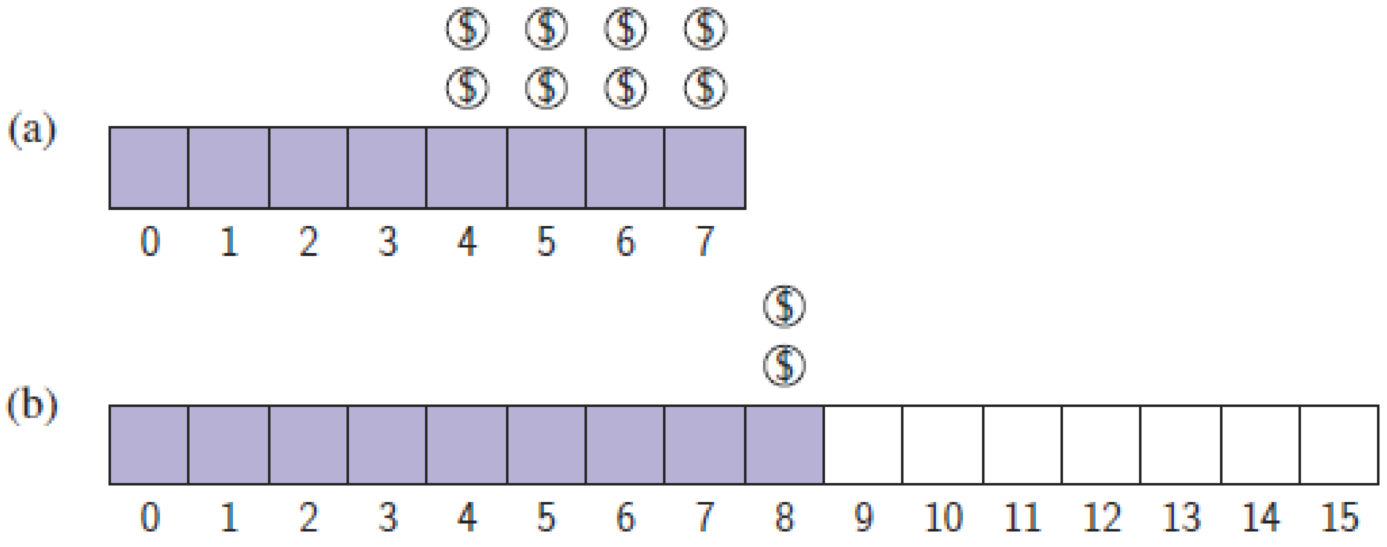


Figure 6.3: A series of push operations on a vector: (a) an 8-cell array is full, with two cyber-dollars "stored" at cells 4 through 7; (b) a push operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table; inserting the new element is paid for by one of the cyber-dollars charged to the push operation; and two cyber-dollars profited are stored at cell 8.

- **Let *V* be a vector implemented by means of an extendable array *A*, as described above. The total time to perform a series of *n* push operations in *V*, starting from *V* being empty and *A* having size $N = 1$, is $O(n)$.**

- Note that the previous overflow occurred when the number of elements became larger than 2^{i-1} for the first time, and thus the cyber-dollars stored in cells 2^{i-1} through $2i - 1$ were not previously spent.
- Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for.
- That is, we can pay for the execution of n push operations using $3n$ cyber-dollars.

STL Vectors

- The Standard Template Library provides C++ programmers a number of useful built-in classes and algorithms.
- The classes provided by the STL are organized in various groups. One of the most important groups is the set of classes called containers.
- A ***container*** is a data structure that stores any collection of elements. We assume that the elements of a container can be arranged in a linear order.
  - Eg: Vectors, Lists, Stacks, Queues

STL Vectors

- The definition of class vector is given in the system include file named “vector.”
- The vector class is part of the std namespace, so it is necessary either to use “std::vector” or to provide an appropriate using statement.
- The vector class is templated with the class of the individual elements.
- The code fragment below declares a vector containing 100 integers.
- We refer to the type of individual elements as the vector’s ***base type***. Each element is initialized to the base type’s default value, which for integers is zero.

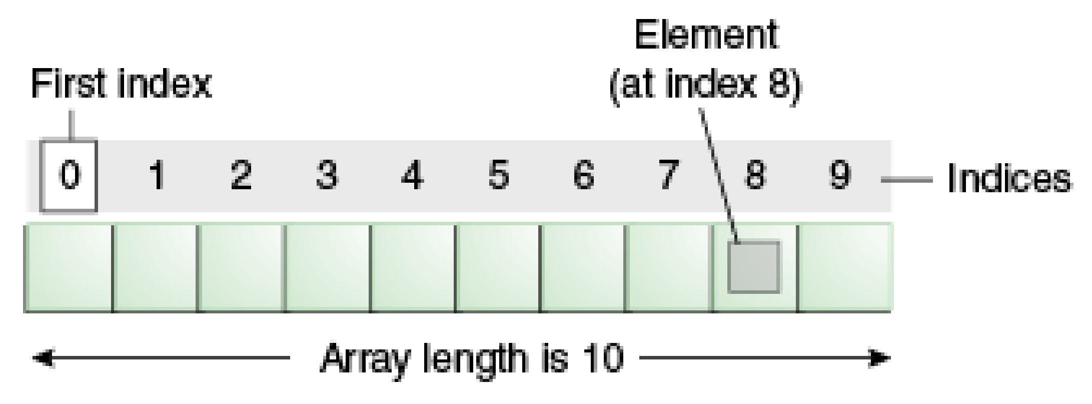
```
#include <vector>                    // provides definition of vector
using std::vector;                   // make vector accessible

vector<int> myVector(100);           // a vector with 100 integers
```

STL Vectors

- STL vector objects behave in many respects like standard C++ arrays, but they provide many additional features.
  - As with arrays, individual elements of a vector object can be indexed using the usual index operator (“[ ]”). Elements can also be accessed by a member function called at.
    - The advantage of at member function is that it performs range checking and generates an error exception if the index is out of bounds.
  - Unlike C++ arrays, STL vectors can be dynamically resized, and new elements may be efficiently appended or removed from the end of an array.

STL Vectors

- When an STL vector of class objects is destroyed, it automatically invokes the destructor for each of its elements. (With C++ arrays, it is the obligation of the programmer to do this explicitly.)
- STL vectors provide a number of useful functions that operate on entire vectors, not just on individual elements. This includes, for example, the ability to copy all or part of one vector to another, the ability to compare the contents of two arrays, and the ability to insert and erase multiple elements.



Member Functions of Vector Class

- Let V be declared to be an STL vector of some base type, and let e denote a single object of this same base type.
- vector(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.
- size(): Return the number of elements in V.
- empty(): Return true if V is empty and false otherwise.
- resize(n): Resize V, so that it has space for n elements.
- reserve(n): Request that the allocated storage space be large enough to hold n elements.

Member Functions of Vector Class

- operator[i]: Return a reference to the i^{th} element of V.
- at(i): Same as $V[i]$, but throw an out of range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V$.size().
- front(): Return a reference to the first element of V.
- back(): Return a reference to the last element of V.
- push_back(e): Append a copy of the element e to the end of V, thus increasing its size by one.
- pop_back(): Remove the last element of V, thus reducing its size by one.

STL Vectors

- When the base type of an STL vector is class, all copying of elements (for example, in push_back) is performed by invoking the class's copy constructor.
- Also, when elements are destroyed the class's destructor is invoked on each deleted element.
- STL vectors are expandable—when the current array space is exhausted, its storage size is increased.
- STL vector also supports functions for inserting elements at arbitrary positions within the vector, and for removing arbitrary elements of the vector.

ArrayVector Class and STL Vectors

- There are both similarities and differences between our ArrayVector class and the STL vector class.
- One difference is that the STL constructor allows for an arbitrary number of initial elements, whereas our ArrayVect constructor always starts with an empty vector.
- The STL vector functions V.front() and V.back() are equivalent to our functions $V[0]$ and $V[n-1]$, respectively, where n is equal to V.size().
- The STL vector functions V.push_back(e) and V.pop_back() are equivalent to our ArrayVect functions V.insert(n,e) and V.remove($n-1$), respectively.

Node-Based Operations and Iterators

- Let L be a (singly or doubly) linked list. We would like to define functions for L that take nodes of the list as parameters and provide nodes as return types.
- Such functions could provide significant speedups over index-based functions, because finding the index of an element in a linked list requires searching through the list incrementally from its beginning or end.
- For instance, a hypothetical function remove(v) that removes the element of L stored at node v of the list.
  - Using a node as a parameter allows us to remove an element in $O(1)$ time by simply going directly to the place where that node is stored and then "linking out" this node through an update of the *next* and *prev* links of its neighbors.

Node-Based Operations and Iterators

- To abstract and unify the different ways of storing elements in the various implementations of a list, we introduce a data type that abstracts the notion of the relative position or place of an element within a list. Such an object might naturally be called a ***position***.
- Because we want this object not only to access individual elements of a list, but also to move around in order to enumerate all the elements of a list, we call it ***iterator*** (similar to convention used in the C++ STL).

Position

- A ***position*** is defined to be an abstract data type that is associated with a particular container and which supports the following function.
  - element(): Return a reference to the element stored at this position.
- Overload the dereferencing operator ("*"), so that, given a position variable p, the associated element can be accessed by *p, rather than p.element().
- This can be used both for accessing and modifying the element's value.

- A position is always defined in a ***relative*** manner, that is, in terms of its neighbors. A position q is "after" some position p and "before" some position r.
- A position q, which is associated with some element e in a container, does not change, even if the index of e changes in the container, unless we explicitly remove e.
- If the associated node is removed, we say that q is ***invalidated***.
- Moreover, the position q does not change even if we replace or swap the element e stored at q with another element.

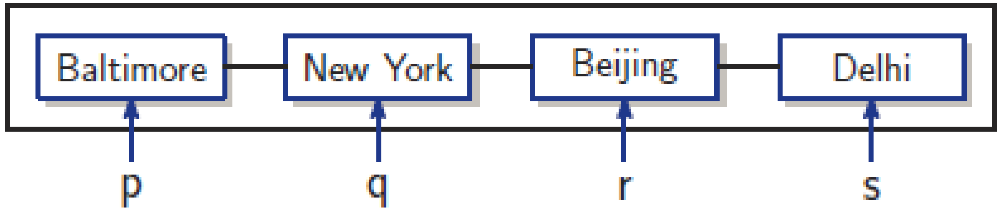


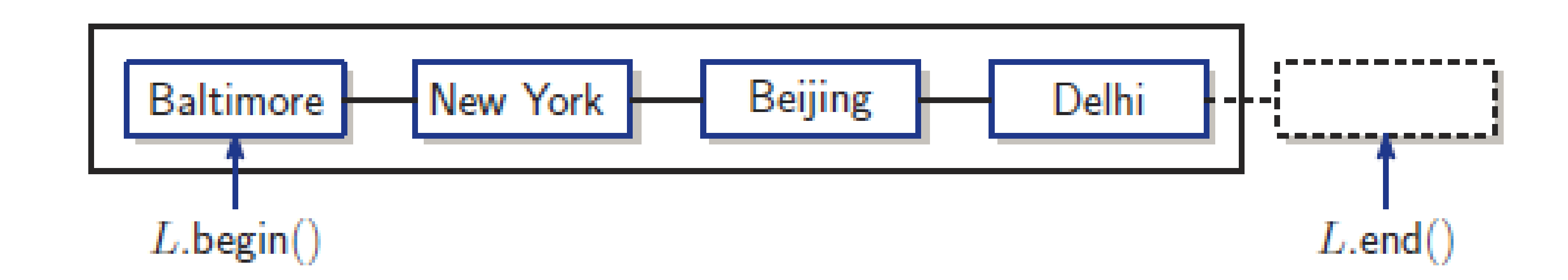
Figure 6.4: A list container. The positions in the current order are p, q, r, and s.

Iterators

- An iterator is an extension of a position. It supports the ability to access a node's element, but it also provides the ability to navigate forwards (and possibly backwards) through the container.
- There are a number of ways in which to define an ADT for an iterator object. Given an iterator object *p*,
  - Define an operation *p*.`next()`, which returns an iterator that refers to the node just after *p* in the container.
  - Overloading the increment operator ("`++`").

Iterators

- Each container provides two special iterator values, ***begin*** and ***end***. The beginning iterator refers to the first position of the container. We think of the ending iterator as referring to an imaginary position that lies ***just after*** the last node of the container.
- With a container object L, the operation L.begin() returns an instance of the beginning iterator for L, and the operation L.end() returns an instance of the ending iterator.

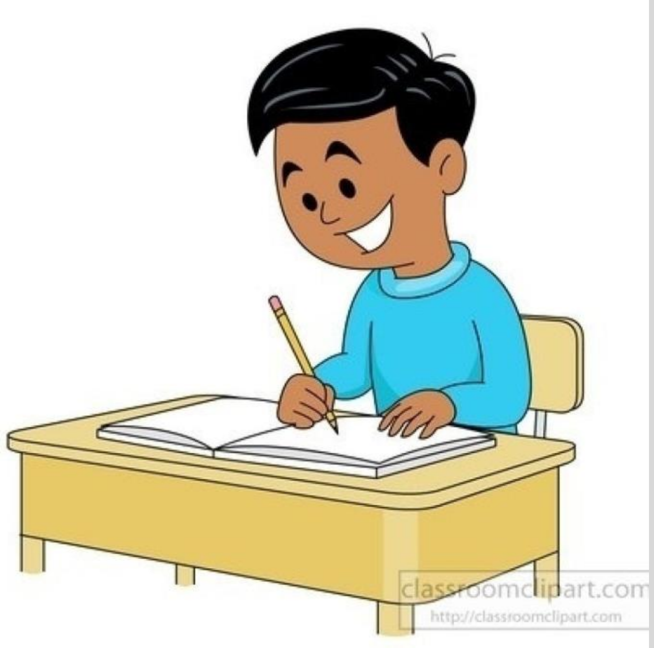


List ADT Functions

- begin(): Return an iterator referring to the first element of L; same as end() if L is empty.
- end(): Return an iterator referring to an imaginary element just after the last element of L.
- insertFront(e): Insert a new element e into L as the first element.
- insertBack(e): Insert a new element e into L as the last element.
- insert(p,e): Insert a new element e into L before position p in L.

List ADT Functions

- eraseFront(): Remove the first element of L.
- eraseBack(): Remove the last element of L.
- erase(p): Remove from L the element at position p; invalidates p as a position.
- The functions insertFront(e) and insertBack(e) are provided as a convenience, since they are equivalent to insert(L.begin(),e) and insert(L.end(),e), respectively.
- Similarly, eraseFront and eraseBack can be performed by the more general function erase.



List Operations

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 Implementation

- Before defining the class, NodeList, we define two important structures.
- The first represents a node of the list and the other represents an iterator for the list. Both of these objects are defined as nested classes within NodeList.
- Since users of the class access nodes exclusively through iterators, the node is declared a private member of NodeList, and the iterator is a public member.

Node Implementation

```
struct Node {                    // a node of the list
  Elem elem;                     // element value
  Node* prev;                    // previous in list
  Node* next;                    // next in list
};
```

Code Fragment 6.6: The declaration of a node of a doubly linked list.

Iterator Implementation

```
class Iterator {                              // an iterator for the list
public:
  Elem& operator*();                          // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                     // move to next position
  Iterator& operator--();                     // move to previous position
  friend class NodeList;                      // give NodeList access
private:
  Node* v;                                    // pointer to the node
  Iterator(Node* u);                          // create from node
};
```

Iterator

◦ To users of class NodeList, it can be accessed by the qualified type name NodeList::Iterator. Its definition, is placed in the public part of NodeList.

◦ An element associated with an iterator can be accessed by overloading the dereferencing operator (“*”).

◦ In order to make it possible to compare iterator objects, we overload the equality and inequality operators (“==” and “!=”).

◦ We provide the ability to move forward or backward in the list by providing the increment and decrement operators (“++” and “– –”).

Iterator

- NodeList is declared to be a friend, so that it may access the private members of Iterator.
- The private data member consists of a pointer *v* to the associated node of the list.
- A private constructor is provided, which initializes the node pointer.
  - The constructor is private so that only NodeList is allowed to create new iterators.

```
NodeList::Iterator::Iterator(Node* u)          // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()          // reference to the element
  { return v->elem; }
                                               // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                               // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                               // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

Iterator

- Observe that the increment and decrement operators not only update the position, but they also return a reference to the updated position.
- This makes it possible to use the result of the increment operation, as in "$q = {+}{+}p$."

Class NodeList Definition

```
typedef int Elem;                                      // list base element type
class NodeList {                                       // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                          // default constructor
  int size() const;                                    // list size
  bool empty() const;                                  // is the list empty?
  Iterator begin() const;                              // beginning position
  Iterator end() const;                                // (just beyond) last position
  void insertFront(const Elem& e);                     // insert at front
  void insertBack(const Elem& e);                      // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
```

Class NodeList Definition

```
  void eraseFront();                              // remove first
  void eraseBack();                               // remove last
  void erase(const Iterator& p);                  // remove p
  // housekeeping functions omitted...
private:                                          // data members
  int     n;                                      // number of items
  Node*   header;                                 // head-of-list sentinel
  Node*   trailer;                                // tail-of-list sentinel
};
```

Code Fragment 6.9: Class NodeList realizing the C++-based list ADT.

Class NodeList

- The class declaration begins by inserting the `Node` and `Iterator` definitions as discussed previously.
- This is followed by the public members, that consist of a simple default constructor and the members of the list ADT.
- We have omitted the standard housekeeping functions from our class definition. These include the class destructor, a copy constructor, and an assignment operator.
- The private data members include pointers to the header and trailer sentinel nodes.
- In order to implement the function `size` efficiently, we also provide a variable *n*, which stores the number of elements in the list.

```
NodeList::NodeList() {                          // constructor
  n = 0;                                        // initially empty
  header = new Node;                            // create sentinels
  trailer = new Node;
  header->next = trailer;                       // have them point to each other
  trailer->prev = header;
}

int NodeList::size() const                      // list size
  { return n; }

bool NodeList::empty() const                    // is the list empty?
  { return (n == 0); }

NodeList::Iterator NodeList::begin() const      // begin position is first item
  { return Iterator(header->next); }

NodeList::Iterator NodeList::end() const        // end position is just beyond last
  { return Iterator(trailer); }
```

- The function begin returns the position just following the header sentinel.
- The function end returns the position of the trailer. As desired, this is the position following the last element of the list.
- In both cases, we are invoking the private constructor declared within class Iterator. We are allowed to do so because NodeList is a friend of Iterator.

Insert Functions

```
                                               // insert e before p
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
  Node* w = p.v;                               // p's node
  Node* u = w->prev;                           // p's predecessor
  Node* v = new Node;                          // new node to insert
  v->elem = e;
  v->next = w; w->prev = v;                    // link in v before w
  v->prev = u; u->next = v;                    // link in v after u
  n++;
}

void NodeList::insertFront(const Elem& e)      // insert at front
  { insert(begin(), e); }

void NodeList::insertBack(const Elem& e)       // insert at rear
  { insert(end(), e); }
```

- Let w be a pointer to p's node, let u be a pointer to p's predecessor.
- We create a new node v, and link it before w and after u.
- Finally, we increment n to indicate that there is one additional element in the list.
- The function insertFront invokes insert on the beginning of the list, and the function insertBack invokes insert on the list's trailer.

Erase Functions

```
void NodeList::erase(const Iterator& p) {    // remove p
  Node* v = p.v;                              // node to remove
  Node* w = v->next;                          // successor
  Node* u = v->prev;                          // predecessor
  u->next = w; w->prev = u;                   // unlink p
  delete v;                                   // delete this node
  n--;                                        // one fewer element
}

void NodeList::eraseFront()                   // remove first
  { erase(begin()); }

void NodeList::eraseBack()                    // remove last
  { erase(--end()); }
```

- Let *v* be a pointer to the node to be deleted, and let *w* be its successor and *u* be its predecessor.
- We unlink *v* by linking *u* and *w* to each other. Once *v* has been unlinked from the list, we need to return its allocated storage to the system in order to avoid any memory leaks.
- We decrement the number of elements in the list.

Space & Time Complexity

- All of the operations of the list ADT run in time $O(1)$.
- The only exceptions to this are the omitted housekeeping functions, the destructor, copy constructor, and assignment operator. They require $O(n)$ time, where n is the number of elements in the list.
- The space used by the data structure is proportional to the number of elements in the list.

STL Containers & Iterators

The STL provides a variety of different container classes.

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

STL Containers & Iterators

- Different containers organize their elements in different ways, and hence support different methods for accessing individual elements.
- STL iterators provide a relatively uniform method for accessing and enumerating the elements stored in containers.
- A simple function that sums the elements of an STL vector, denoted by V

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

STL Containers & Iterators

- This particular method of iterating through the elements of a vector is quiet familiar.
- Unfortunately, this method would not be applicable to other types of containers, because it relies on the fact that the elements of a vector can be accessed efficiently through indexing.
- This is not true for all containers, such as lists. What we desire is a uniform mechanism for accessing elements.
- Every STL container class defines a special associated class called an ***iterator***.
- If *p* is an iterator that refers to some position within a container, then **p* yields a reference to the associated element.

STL Containers & Iterators

- Advancing to the next element of the container is done by incrementing the iterator.
- For example, either ++*p* or *p*++ advances *p* to point to the next element of the container. The former returns the updated value of the iterator, and the latter returns its original value.
- Each STL container class provides two member functions, begin and end, each of which returns an iterator for this container.
- The first returns an iterator that points to the first element of the container, and the second returns an iterator that can be thought of as pointing to an imaginary element ***just beyond*** the last element of the container.

Using Iterators

- Suppose, for example, that *C* is of type vector<int>, that is, it is an STL list of integers. The associated iterator type is denoted "vector<int>::iterator."
- In general, if *C* is an STL container of some type cont and the base type is of type base, then the iterator type would be denoted "cont<base>::iterator."

```
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;
}
```

Using Iterators

- Most STL containers (including lists, sets, and maps) provide the ability to move not only forwards, but backwards as well. For such containers the decrement operators $--p$ and $p--$ are also defined for their iterators. This is called a ***bidirectional iterator***.
- A few STL containers (including vectors and deques) support the additional feature of allowing the addition and subtraction of an integer. For example, for such an iterator, p, the value $p+3$ references the element three positions after p in the container. This is called a ***random-access iterator***.

Using Iterators

- It is up to the programmer to be sure that an iterator points to a valid element of the container before attempting to dereference it.
- Attempting to dereference an invalid iterator can result in your program aborting.
- Iterators can be ***invalid*** for various reasons. For example, an iterator becomes invalid if the position that it refers to is deleted.

STL Queue

- As with the STL vector, the class queue is part of the std namespace, so it is necessary either to use “std::queue” or to provide an appropriate “using” statement.
- The queue class is templated with the base type of the individual elements. For example, the code fragment below declares a queue of floats.

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

- An STL queue dynamically resizes itself as new elements are added.

STL Queue

- Let *q* be declared to be an STL queue, and let *e* denote a single object whose type is the same as the base type of the queue. (For example, *q* is a queue of floats, and *e* is a float.)
- 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

- The result of applying any of the operations front, back, or pop to an empty STL queue is undefined. Though, no exception is thrown, but it may very likely result in the program aborting.
- It is up to the programmer to be sure that no such illegal accesses are attempted.

STL Deque

- The Standard Template Library provides an implementation of a deque.
- First, we need to include the definition file “deque.” Since it is a member of the std namespace, we need to either preface each usage “std::deque” or provide an appropriate “using” statement.
- The deque class is templated with the base type of the individual elements.
- For example, the code fragment below declares a deque of strings.

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

STL Deque

- An STL deque dynamically resizes itself as new elements are added.
- 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.

STL Deque

- The result of applying any of the operations front, back, push_front, or push_back to an empty STL queue is undefined. Thus, no exception is thrown, but the program may abort.

STL Lists

- As with the STL vector, the list class is a member of the std namespace, it is necessary either to preface references to it with the namespace resolution operator, as in "std::list", or to provide an appropriate using statement.
- The list class is templated with the ***base type*** of the individual elements. For example, the code fragment below declares a list of floats.
- By default, the initial list is empty.

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

List ADT

- Let L be declared to be an STL list of some base type, and let x denote a single object of this same base type. (For example, L is a list of integers, and e is an integer.)
- list(n): Construct a list with n elements; if no argument list is given, an empty list is created.
- size(): Return the number of elements in L.
- empty(): Return true if L is empty and false otherwise.
- front(): Return a reference to the first element of L.
- back(): Return a reference to the last element of L.

List ADT

- push_front(e): Insert a copy of e at the beginning of L.
- push_back(e): Insert a copy of e at the end of L.
- pop_front(): Remove the fist element of L.
- pop_back(): Remove the last element of L.

Sequences

- It is an abstract data type that generalizes the vector and list ADTs.
- This ADT therefore provides access to its elements using both indices and positions, and is a versatile data structure for a wide variety of applications.
- The interface consists of the operations of the list ADT, plus the following two “bridging” functions, which provide connections between indices and positions.
- atIndex(i): Return the position of the element at index i.
- indexOf(p): Return the index of the element at position p.

Sequences

- We discuss the definition of a class NodeSequence, which implements the sequence ADT.
- A sequence extends the definition of a list, thus we have inherited NodeSequence class by extending the NodeList class.
- The NodeSequence class have access to all the members of the NodeList class, including its nested class, NodeList::Iterator.

```
class NodeSequence : public NodeList {
public:
  Iterator atIndex(int i) const;            // get position from index
  int indexOf(const Iterator& p) const;     // get index from position
};
```

Sequences

```
                                        // get position from index
NodeSequence::Iterator NodeSequence::atIndex(int i) const {
  Iterator p = begin();
  for (int j = 0; j < i; j++) ++p;
  return p;
}
                                        // get index from position
int NodeSequence::indexOf(const Iterator& p) const {
  Iterator q = begin();
  int j = 0;
  while (q != p) {                      // until finding p
    ++q; ++j;                           // advance and count hops
  }
  return j;
}
```

- The function `atIndex(`i`)` hops i positions to the right, starting at the beginning, and returns the resulting position.
- The function `indexOf` hops through the list until finding the position that matches the given position p.
- Observe that the conditional "`q != p`" uses the overloaded comparison operator for positions defined in Iterator class.

Sequences

- Appropriate exception should be thrown if
  - If `atIndex` gets the argument i that does not lie in the range from 0 to $n-1$, where n is the size of the sequence.
  - The function `indexOf` should check that it does not run past the end of the sequence.

Sequences

- The worst-case running times of both of the functions `atIndex` and `indexOf` are $O(n)$, where n is the size of the list.
- Although this is not very efficient, we may take consolation in the fact that all the other operations of the list ADT run in time $O(1)$.
- A natural alternative approach would be to implement the sequence ADT using an array. Although we could now provide very efficient implementations of `atIndex` and `indexOf`, the insertion and removal operations of the list ADT would now require $O(n)$ time.
- Thus, neither solution is perfect under all circumstances.

Implementing a Sequence with Array

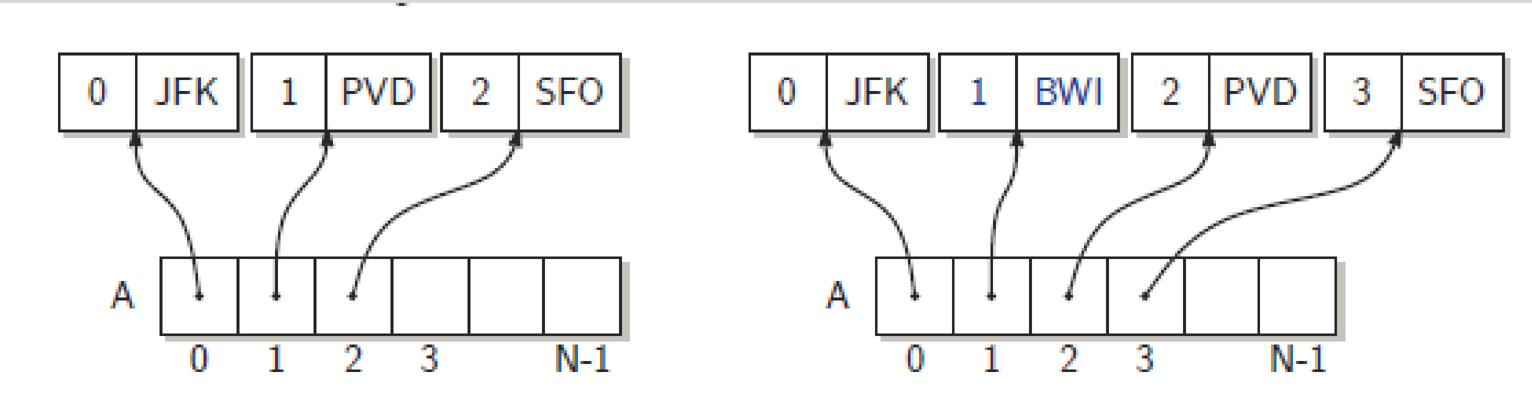
- Suppose we want to implement a sequence S by storing each element e of S in a cell $A[i]$ of an array A.
- The position object p can be defined to hold an index i and a reference to array A, as member variables.
- Then function element(p) can be implemented by simply returning a reference to $A[i]$.
- A major drawback with this approach, however, is that the cells in A have no way to reference their corresponding positions.

Implementing a Sequence with Array

- For example, after performing an `insertFront` operation, the elements have been shifted to new positions, but we have no way of informing the existing positions for *S* that the associated positions of their elements have changed.
- Thus, we need a different approach.

Implementing a Sequence with Array

- Instead of storing the elements of S in array A, we store a pointer to a new kind of position object in each cell of A.
- Each new position object p stores a pair consisting of the index i and the element e associated with p.
- We can easily scan through the array to update the i value associated with each position whose rank changes as the result of an insertion or deletion.



Implementing a Sequence with Array

- In this array-based implementation of a sequence, the functions insertFront, insert, and erase take $O(n)$ time because we have to shift position objects to make room for the new position or to fill in the hole created by the removal of the old position.
- All the other position-based functions take $O(1)$ time.
- Note that we can use an array in a circular fashion, as we did for implementing a queue. We can then perform functions insert-Front in $O(1)$ time. Note that functions insert and erase still take $O(n)$ time.

Running Times

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	$O(n)$	$O(1)$

Space Usage

- An array requires $O(N)$ space, where N is the size of the array (unless we utilize an extendable array), while a doubly linked list uses $O(n)$ space, where n is the number of elements in the sequence.
- Since n is less than or equal to N, this implies that the asymptotic space usage of a linked-list implementation is superior to that of a fixed-size array, although there is a small constant factor overhead that is larger for linked lists, since arrays do not need links to maintain the ordering of their cells.