3.1 Abstract Data Type

An *Abstract Data Type* (ADT) is a set of objects with a set of associated operations. An ADT is

- purely abstract, with no requirements on the implementation
- a generalization of primitive types (integers, reals, booleans)
- implemented as a C++ class with details hidden

ADT’s include the following.

- List
- Stack
- Queue and Deque (double-ended queue)
- Tree
- Set and Multiset
- Map (associative array) and Multimap
- Priority queue
3.2 List ADT

A list is an ordered sequence of objects, $A_0, A_1, \ldots, A_{N-1}$, each with a position, a predecessor (except for the first), and a successor (except the last). The operations include

- printList
- makeEmpty or clear
- find (position of first occurrence, if present)
- insert (at some position)
- remove or erase (from some position)
- findKth (return value at some specified position)
- next (return pointer to successor of element at specified location)
- previous (return pointer to predecessor of element at some specified location)

Note that positions are pointers in a linked list, indexes in an array.
Methods for implementing a list include the following:

1. **Array**: growth is restricted because the size is fixed.

2. **Vector**: allows doubling of size but has expensive insert and remove, especially at position 0.

3. **Singly linked list (with a link to the front)**: reduces the cost of insert and remove at a known position by allowing nonadjacent nodes, but has a problem with \texttt{findKth}.

4. **Doubly linked list**

A singly linked list requires an $O(K)$ traversal for \texttt{findKth}, but a sequence of calls sorted on $K$ could be performed efficiently. An insertion at the back could be made efficient by keeping links (pointers) to both ends. What about removing the last node?
Figures 3.2-3.4

Figure 3.2 Deletion from a linked list

Figure 3.3 Insertion into a linked list

Figure 3.4 A doubly linked list
Templates provide software re-use through generic programming by allowing type-independent algorithms and overloaded functions.

STL containers (collections) for the list ADT:

1. vector: a growable array (indexed in constant time)
2. list: a doubly linked list (not easily indexed, but allows cheap insertion and removal)

Both containers are class templates instantiated with the type of element. Note that neither is efficient for searches.

The following methods are common to all STL containers:

- int size(): returns number of elements in container
- void clear(): removes all elements from container
- bool empty(): returns true if empty, false otherwise
The following methods are supported by both vector and list:

- **void push_back(const Object & x)**: adds x to end of list
- **void pop_back()**: removes object at end of list
- **const Object & back() const**: returns object at end of list (accessor)
- **Object & back()**: returns object at end of list (mutator)
- **const Object & front() const**: returns object at front (accessor)
- **Object & front()**: returns object at front (mutator)
The following methods are supported only by list:

- `void push_front(const Object & x)`: adds `x` to front of list
- `void pop_front()`: removes object at front of list

The following methods are supported only by vector:

- `Object & operator[](int idx)` (No bounds checking)
- `const Object & operator[](int idx) const` (No bounds checking)
- `Object & at(int idx)` (Bounds checking)
- `const Object & at(int idx) const` (Bounds checking)
- `int capacity() const`
- `void resize(int newCapacity)`
3.3.1 Iterators

An **iterator** is a nested type (class) used by specializations of STL containers to represent a position in the container. Iterators are designed to mimic pointers.

Every STL container provides the following methods:

- `iterator begin()`: returns iterator representing first item
- `const_iterator begin() const` (accessor)
- `iterator end()`: returns iterator representing end marker.
- `const_iterator end() const` (accessor)

The most common iterator methods are the following:

- `operator=`
- `copy constructor`
- `itr++`, `++itr` (and possibly `itr--`, `--itr`)
- `*itr`
- `operator==`, `operator!=`
Iterators continued

Example

```cpp
vector<int> v;
v.push_back(1);
v.push_back(2);
for (vector<int>::iterator itr = v.begin();
    itr != v.end(); ++itr)
    cout << *itr << endl;
```

The following list ADT methods (used in the text implementation of List) require iterators:

- **iterator insert(iterator pos, const Object & x)**: insert prior to pos; return position of inserted item
- **iterator erase(iterator pos)**: remove object at position pos; return position of element that followed pos prior to the call; pos becomes stale
- **iterator erase(iterator start, iterator end)**: remove elements from start up to, but not including end
Figure 3.5

```cpp
template <typename Container>
void removeEveryOtherItem( Container & lst )
{
    typename Container::iterator itr = lst.begin();
    while( itr != lst.end() )
    {
        itr = lst.erase( itr );
        if( itr != lst.end() )
            ++itr;
    }
}
```

**Figure 3.5** Using iterators to remove every other item in a List (either a vector or list). Efficient for a list, but not for a vector.
3.3.3 const_iterators

Like pointers, iterators may be dereferenced, and *itr may appear on either side of an assignment operator. The following generic type-independent code example demonstrates the benefit.

```cpp
template <typename Container, typename Object>
void change(Container & c, const Object & newValue)
{
    typename Container::iterator itr = c.begin();
    while (itr != c.end())
        *itr++ = newValue;
}
```

Note that keyword typename has two different uses: templates and qualified type names that depend on a template parameter. It tells the compiler that Container::iterator is a type name rather than a data field or method. The binary scope resolution operator :: is necessary because an iterator is a nested type with different implementations in different containers.
Problem: When a container is passed to a function by constant reference, how does the compiler prevent it from being changed by one of its mutators?

Solution A `const_iterator`, whose `operator*` member returns a constant reference, is provided by every STL container, and is required to reference a constant container.

```cpp
void print(const list<int> & lst, ostream & out = cout) {
    list<int>::const_iterator itr = lst.begin();
    while (itr != lst.end()) {
        out << *itr << endl;
        *itr = 0;  //compile error: requires iterator
        itr++;
    }
}
```

Overloaded functions `begin` and `end` return `const iterators`. Declaring `itr` to be an iterator would produce a compile error.
template <typename Container>
void printCollection( const Container & c, ostream & out = cout )
{
    if( c.empty() )
        out << "(empty)";
    else
    {
        typename Container::const_iterator itr = c.begin();
        out << "[ " << *itr++;
        // Print first item
        while( itr != c.end() )
            out << ", " << *itr++;
        out << "]" << endl;
    }
}

Figure 3.6 Printing any container
Figure 3.7

```cpp
template<typename Object>
class Vector {
  public:
    explicit Vector( int initSize = 0 )
      : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
        { objects = new Object[ theCapacity ]; }
    Vector( const Vector & rhs ) : objects( NULL )
      { operator=( rhs ); }
  ~Vector() {
    delete [ ] objects; }

  const Vector & operator= ( const Vector & rhs )
  {
    if( this != &rhs )
    {
      delete [ ] objects;
      theSize = rhs.size();
      theCapacity = rhs.theCapacity;
      objects = new Object[ capacity( ) ];
      for( int k = 0; k < size(); k++ )
        objects[ k ] = rhs.objects[ k ];
    }
    return *this;
  }

  void resize( int newSize )
  {
    if( newSize > theCapacity )
      reserve( newSize * 2 + 1 );
    theSize = newSize;
  }

  void reserve( int newCapacity )
  {
    if( newCapacity < theSize )
      return;
    Object *oldArray = objects;
    objects = new Object[ newCapacity ];
    for( int k = 0; k < theSize; k++ )
      objects[ k ] = oldArray[ k ];
    theCapacity = newCapacity;
    delete [ ] oldArray;
  }
```
50   Object & operator[]( int index )
51   { return objects[ index ]; }  
52   const Object & operator[]( int index ) const
53   { return objects[ index ]; }  
54
55   bool empty() const
56   { return size() == 0; }  
57   int size() const
58   { return theSize; }  
59   int capacity() const
60   { return theCapacity; }  
61
62   void push_back( const Object & x )
63   {  
64       if( theSize == theCapacity )
65           reserve( 2 * theCapacity + 1 );
66       objects[ theSize++ ] = x;  
67   }
68
69   void pop_back( )
70   { theSize--; }  
71
72   const Object & back() const
73   { return objects[ theSize - 1 ]; }  
74
75   typedef Object * iterator;
76   typedef const Object * const_iterator;
77
78   iterator begin( )
79   { return &objects[ 0 ]; }  
80   const_iterator begin( ) const
81   { return &objects[ 0 ]; }  
82   iterator end( )
83   { return &objects[ size() ]; }  
84   const_iterator end( ) const
85   { return &objects[ size() ]; }  
86
87   enum { SPARE_CAPACITY = 16 };  
88
89   private:
90   int theSize;
91   int theCapacity;
92   Object * objects;
93   };

Figure 3.8 vector class (Part 2 of 2)
3.4 Implementation of vector

The text class Vector is a simplified implementation of the STL vector container.

- It needs error checking.

- `operator=` should only free and reallocate memory if necessary: if `capacity() < rhs.capacity()`.
  However, if `capacity() > rhs.capacity()` the assignment would not produce an exact duplicate.

- Note the postfix `++` operator in `push_back`:
  `objects[ theSize++ ] = x;`.

- `pop_back` and `back` should both test for `theSize > 0` before decrementing or referencing `objects[ theSize - 1 ]`.

- `iterator` and `const_iterator` are aliases (typedef) for pointers (which have all the same operators required of an iterator).
  However, neither `++itr` nor `*itr` will signal an error with a bad pointer. Error checks require that iterators be implemented as nested classes (as in the case of List).
3.5 Implementation of list

The text class `List` is a simplified implementation of STL container `list`. It contains links (pointers) to **sentinel nodes** — a header node and a tail node which eliminate special cases such as removing the first node.

![Doubly linked list with header and tail nodes](image)

**Figure 3.9** A doubly linked list with header and tail nodes

![Empty doubly linked list with header and tail nodes](image)

**Figure 3.10** An empty doubly linked list with header and tail nodes
```cpp
1  template <typename Object>
2  class List
3  {
4      private:
5          struct Node
6             { /* See Figure 3.13 */; }
7
8      public:
9          class const_iterator
10             { /* See Figure 3.14 */; }
11
12          class iterator : public const_iterator
13             { /* See Figure 3.15 */; }
14
15      public:
16          List()  
17              { /* See Figure 3.16 */ }
18          List( const List & rhs )
19              { /* See Figure 3.16 */ }
20          ~List()  
21              { /* See Figure 3.16 */ }
22          const List & operator= ( const List & rhs )
23              { /* See Figure 3.16 */ }
24
25          iterator begin()  
26              { return iterator( head->next ); } 
27          const_iterator begin() const
28              { return const_iterator( head->next ); } 
29          iterator end()  
30              { return iterator( tail ); } 
31          const_iterator end() const
32              { return const_iterator( tail ); }
33
34          int size() const
35              { return theSize; }
36          bool empty() const
37              { return size() == 0; }
38
39          void clear()  
40          {
41              while( !empty() )
42                  pop_front();
43          }
```
```cpp
44   Object & front( )
45   { return *begin( ); }
46   const Object & front( ) const
47   { return *begin( ); }
48   Object & back( )
49   { return *--end( ); }
50   const Object & back( ) const
51   { return *--end( ); }
52   void push_front( const Object & x )
53   { insert( begin( ), x ); }
54   void push_back( const Object & x )
55   { insert( end( ), x ); }
56   void pop_front( )
57   { erase( begin( ) ); }
58   void pop_back( )
59   { erase( --end( ) ); }
60
61   iterator insert( iterator itr, const Object & x )
62   { /* See Figure 3.18 */ }
63
64   iterator erase( iterator itr )
65   { /* See Figure 3.20 */ }
66   iterator erase( iterator start, iterator end )
67   { /* See Figure 3.20 */ }
68
69   private:
70      int theSize;
71      Node *head;
72      Node *tail;
73
74   void init( )
75   { /* See Figure 3.16 */ }
76   );
```

**Figure 3.12 List class (Part 2 of 2)**
A struct is a class in which members default to public — used for a type that contains mostly data that are accessed directly. Note that Node is a private nested class in List.
class const_iterator
{
    public:
    const_iterator() : current(NULL) 
    
    const Object & operator* ( ) const 
    { return retrieve(); }

    const_iterator & operator++ ( )
    {
        current = current->next;
        return *this;
    }

    const_iterator operator++ ( int )
    {
        const_iterator old = *this;
        ++(*this);
        return old;
    }

    bool operator==( const const_iterator & rhs ) const 
    { return current == rhs.current; }

    bool operator!=( const const_iterator & rhs ) const 
    { return !( *this == rhs ); }

    protected:
    Node *current;

    Object & retrieve( ) const 
    { return current->data; }

    const_iterator( Node *p ) : current( p )
    
    friend class List<Object>;
};

Figure 3.14 Nested const_iterator class for List class
const_iterator class

The const_iterator class contains a pointer to the "current" node and overloaded operators *, ++, ==, and !=. The pointer is protected, rather than private, so that iterator, which is derived from const_iterator, has access to it. The one-parameter constructor is protected so that functions outside of List can't create iterators with arbitrary initial values. Since it is used by List's begin and end members, it must be made accessible by the friend declaration on line 37. The declaration actually makes every member function of class List<Object> a friend of const_iterator, giving them access to all data in const_iterator. Note that the postfix operator++ on line 16 has an unused int argument to change the signature from the prefix operator. Also, note that *itr++ means dereference itr, then increment itr: *itr++ ≠ *(itr++) and *itr++ ≠ (*itr)+++. Hence the postfix operator++ must advance pointer current but return the old iterator. Lines 24 and 26 indicate an arbitrary choice for the == test.
class iterator : public const_iterator
{
    public:
    iterator( )
    {
    }

    Object & operator* ( )
    { return retrieve( ); }
    const Object & operator* ( ) const
    { return const_iterator::operator*( ); }

    iterator & operator++ ( )
    {
        current = current->next;
        return *this;
    }

    iterator operator++ ( int )
    {
        iterator old = *this;
        ++( *this );
        return old;
    }

protected:
    iterator( Node *p ) : const_iterator( p )
    {
    }

friend class List<Object>;
};

**Figure 3.15** Nested iterator class for List class
Class iterator is a nested class derived from base class const_iterator, and inherits all members but redefines operator++ (with a changed return type) and operator* (by adding a mutator). The added mutator hides the inherited accessor, requiring another accessor.

The syntax 'class iterator : public const_iterator' implies inheritance: iterator is a const_iterator.
Figure 3.16 Constructor, Big-Three, and private init routine for List class

```cpp
    List( )
    { init( ); }

    List( )
    {
        clear( );
        delete head;
        delete tail;
    }

    List( const List & rhs )
    {
        init( );
        *this = rhs;
    }

    const List & operator= ( const List & rhs )
    {
        if( this == &rhs )
            return *this;
        clear( );
        for( const_iterator itr = rhs.begin( ); itr != rhs.end( ); ++itr )
            push_back( *itr );
        return *this;
    }

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
```
**Figures 3.17 and 3.18**

**Figure 3.17** Insertion in a doubly linked list by getting new node and then changing pointers in the order indicated.

The only restriction is actually that 1 and 3 precede 4.

```cpp
1    // Insert x before itr.
2    iterator insert( iterator itr, const Object & x )
3    {
4        Node *p = itr.current;
5        theSize++;
6        return iterator( p->prev = p->prev->next = new Node( x, p->prev, p ) );
7    }
```

**Figure 3.18** insert routine for List class
Figure 3.19 Removing node specified by p from a doubly linked list

```c
// Erase item at itr.
iterator erase( iterator itr )
{
    Node *p = itr.current;
    iterator retVal( p->next );
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    theSize--;
    return retVal;
}

iterator erase( iterator start, iterator end )
{
    for( iterator itr = from; itr != to; )
    {
        itr = erase( itr );
    }
    return to;
}
```

Figure 3.20 erase routine for List class
Note that start and end should be from and to.
Figure 3.21 Revised protected section of \texttt{const\_iterator} that incorporates ability to perform additional error checks

Potential errors in \texttt{erase} and \texttt{insert} include uninitialized iterators and iterators for the wrong list. Testing for the correct list requires that the iterator store a pointer to the \texttt{List} from which it was constructed.
// Insert x before itr.
iterator insert( iterator itr, const Object & x )
{
    itr.isValid();
    if( itr.theList != this )
        throw IteratorMismatchException();

    Node *p = itr.current;
    theSize++;
    return iterator( *this,
                    p->prev = p->prev->next = new Node( x, p->prev, p ) );
}

Figure 3.22 List insert with additional error checks
3.6 The Stack ADT

- A stack is a list with the restriction that insertions and deletions can only occur at the top.
- A stack is a LIFO data structure.
- The fundamental operations are push, pop, top (return top element), size, clear, and empty.
- Pop or top on an empty stack is an error, but exceeding stack space with a push is not an ADT error.
- A stack can be implemented as a vector, list, or something simpler like a singly linked list with the top at the front, or an array with the top at the back (high address), but the simplification over a vector or list is not a significant cost saving. Stack operations are very efficient, often requiring only one machine language instruction.
Figure 3.24 Stack model: only the top element is accessible

array implementation

push(x): topOfStack++; theArray[TopOfStack] = x;
top: return theArray[topOfStack];
pop: topOfStack--; or
pop: return theArray[topOfStack--];
1) Balancing Symbols

Opening and closing symbols in a source code file (begin/end pairs, function call/return, etc) must be balanced. For example, [...(...)...] is legal, but [...(...][...]) is not. A linear-time, one-pass, on-line algorithm is as follows.

```plaintext
clear stack
for each character c in the source file
    if (c == opening symbol) push(c)
    if (c == closing symbol) {
        if (stack is empty) error
        s = pop()
        if (s is not the opening symbol associated with c) error
    }
if (stack is not empty) error
```
2) **Postfix Expression Evaluation**

The *postfix* (reverse Polish notation, RPN) expression 'a b c * +' evaluates to (the *infix* expression) 'a + (b*c)'. There is no ambiguity as there is in 'a + b * c', which is evaluated as (a+b)*c when followed by '=' on a 4-function calculator.

Postfix expression evaluation is implemented with a stack. On an HP calculator, for example, the Enter key is a push, and an arithmetic operator replaces the entry at the top of the stack with the result of applying the operator to the top two entries. For example, 'a b c * +' results in the following sequence of stack entries

\[
\begin{align*}
a & \rightarrow b \\
b & \rightarrow c \\
c & \rightarrow b * c \\
a & \rightarrow a + (b * c)
\end{align*}
\]

No precedence rules are needed.
As another example, ‘a b c + /’ produces the following

\[
\begin{align*}
    a & \rightarrow b \\
    b & \rightarrow b + c \\
    c & \rightarrow a \\
    a & \rightarrow a/(b + c)
\end{align*}
\]

The general rule is that ‘a op b’ is computed by the sequence enter a, enter b, apply the operator.

The Intel math coprocessor (floating-point hardware) uses a stack of 80-bit registers but includes both stack operations and instructions for random access to stack entries (by register designator).

3) **Infix to Postfix Conversion**
A simplified problem is the following. Given a legal infix expression involving only operands, parentheses, and the four operators +, -, *, /, convert it to postfix.
Infix to Postfix Conversion Algorithm

Empty the stack
for each token t in the input stream
  if (t is an operand)
    output t
  else if (t is a right parenthesis)
    pop and output tokens until a left parenthesis
    is popped (but not output)
  else // t is an operator or left parenthesis
    pop and output tokens until one of lower priority
    than t is encountered or a left parenthesis is
    encountered or the stack is empty
    push t
  endif
endfor
pop and output tokens until the stack is empty
Example:

\[ a + b \cdot c + (d \cdot e + f) \cdot g \longrightarrow a \ b \ c \ * \ + \ d \ e \ * \ f \ + \ g \ * \ + \]

Note that in the output stream produced by the algorithm, operators of equal precedence associate from left to right: 
\[ a \cdot b / c \longrightarrow a \ b \ * \ c \ /, \] which is correct, but if we add an exponentiation operator \(^\wedge\), we must change the algorithm so that \(^\wedge\) is associated from right to left: 
\[ a ^ b ^ c \longrightarrow a \ b \ c \ ^ \ ^\wedge \]. For example, \[ 2 ^ 2 ^ 3 = 2^8 \neq 4^3. \]
4) **Function Calls**

A function call and return pair is analogous to a pair of parentheses in that a return is paired with the most recent call (the one at the top of the stack).

A difference is that more information must be stored. A *stack frame* or activation record contains at least the return address of the calling function, and possibly passed parameters, saved register values for the calling procedure, and local (automatic) variables for the called procedure.

The Intel processors have always had three registers (SS, SP, BP) and several instructions associated with the runtime stack in memory (in addition to the floating-point register stack). The stack grows downward from the top of the address space while the heap (used for dynamic memory allocation) grows upward from the bottom.
A bad use of recursion: printing a container

Stack overflow is usually the result of bad recursion; i.e., the normal level of function call nesting is not that deep. The above code is an example of tail recursion: the last line is a recursive call. Since there is no need to save the state of the calling function, the recursion can be replaced by a loop.
/**
 * Print container from start up to but not including end.
 */

template <typename Iterator>
void print( Iterator start, Iterator end, ostream & out = cout )
{
    while( true )
    {
        if( start == end )
            return;

        out << *start++ << endl;  // Print and advance start
    }
}

Figure 3.26 Printing a container without recursion; a compiler might do this (you should not)
**Problem**: Given three poles and a set of \( n \) disks, no two of which are the same size, all on one pole with sizes decreasing from bottom to top, move all the disks to another pole using the rule that only one disk at a time may be moved, and it can only be placed on an empty pole or on top of a larger disk.

**Recursive Solution**:

```c
void hanoi(int n, Stack fromPole, Stack toPole, Stack aux) {
    if (n >= 1) {
        hanoi(n-1, fromPole, aux, toPole);
        Disk d = fromPole.pop();
        toPole.push(d);
        hanoi(n-1, aux, toPole, fromPole);
    }
}
```

**Complexity** \( T(n) = 2T(n-1) + k, T(0) = 0 \Rightarrow T(1) = k, T(2) = 3k, T(3) = 7k, \ldots, T(n) = (2^n-1)k. \)
A queue is a list with the restriction that insertions only occur at the back, and deletions only occur at the front.

A queue is FIFO data structure.

The fundamental operations, in addition to size, clear, and empty, are enqueue (insert at back) and dequeue (delete and return the element at the front).

As with a stack, any implementation is legal, and both the linked list and array or vector implementations give fast $O(1)$ run times for every operation (but not for searching).
### 3.7.2 Array Implementation of Queues

**Data members:** theArray, currentSize, front, back

**enqueue:**

```plaintext
++back;
theArray[back] = x;
++currentSize;
```

**dequeue:**

```plaintext
if (!currentSize) error
retVal = theArray[front];
--currentSize;
++front;
return retVal;
```

**Problem:** Every enqueue operation increases the required array size even though the queue size might remain small.

**Solution:** Use a *circular array* in which the first array element follows the last. For array size $n$ (indices 0 to $n-1$) all increments and decrements are modulo $n$: $\text{back} = (\text{back} + 1) \mod n$ instead of $\text{back}++$, and an empty queue has $\text{back} = (\text{front}-1) \mod n$. Note that currentSize is redundant but, if we don’t store it, we can only store $n-1$ elements because storing $n$ elements would make $\text{back} = (\text{front}-1) \mod n$ indicating an empty queue.
3.7.3 Applications of Queues

- Operating systems schedule jobs (with equal priority) in the order of arrival; e.g., a print queue. Note, however, that canceling a job effectively deletes it from an arbitrary position.
- Simulation of real-world queues such as lines at a ticket counter, or any other first-come first-served scenario requires a queue.
- Servers on a network generally operate in a first-come first-served fashion.
- Calls to a customer support site are usually queued.
- *Queuing Theory* is a branch of mathematics that computes wait times and queue lengths from arrival time probability distributions and processing time probability distributions. In complicated situations this requires a simulation using queues and other data structures.