Lists, Stacks, and Queues

ADT

- Abstract Data Type (ADT):

  a set of objects together with a set of operations.
- ADT does not specify how the set of operations is implemented.
- Allows transparent changes to the implementation.

Lists

- Array implementation
- Linked list implementation
  - Single linked list
  - Doubly linked list
- Basic operations:
  - printList
  - find
  - findKth
  - insert
  - remove

Linked lists: developed by A. Newell, C. Shaw and H. A. Simon at RAND Corporation (1955); data structure for their Information Processing Language (IPL).

LISP's major data structure (1958)

Vector and list in the STL

- Standard Template Library (STL)
- Includes implementation of common data structures
- Data structures are called collections or containers
- Implementation of List ADT:
  - vector: growable array implementation
  - list: doubly linked list implementation

Vector and list: Methods

- **`int size() const`**: returns the number of elements in the container
- **`void clear()`**: removes all elements
- **`bool empty() const`** : true if empty container
- **`void push_back(const Object & x)`**: adds x to the end of list
- **`void pop_back()`**: removes the object at the end of the list
- **`const Object & back() const`**: returns the object at the end of the list
- **`const Object & front() const`**: returns the object at the front of the list

Methods specific to list

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

Methods specific to vector

- **`Object & operator [] (int idx):`** returns the object at index idx in the vector with no bounds-checking
- **`Object & at(int idx)`**: returns the object at index idx in the vector, with bounds-checking
- **`int capacity () const`**: returns the internal capacity of the vector
- **`void reserve (int new Capacity)`** : sets the new capacity

Iterators

- Position in STL is represented by a nested type: **`iterator`**
- Issues:
  - How to get an iterator?
  - What operations the iterators can perform?
  - Which list ADT methods require iterators as parameters?

Getting an iterator

- **`iterator begin( )`** : returns an appropriate iterator representing the first item in the container
- **`iterator end( )`** : returns an appropriate iterator representing the end marker in the container (i.e., the position after the last item)
- Example:

```
for( int i = 0; i != v.size(); ++i)
  cout << v[i] << endl;

for( vector<int>::iterator itr = v.begin(); itr != v.end(); itr.???)
  cout << itr.??? << endl;
```

Iterator Methods

- **`itr++`** and **`++itr`** : advances the iterator to the next location
- **`*itr`**: returns a reference to the object stored at iterator location
- **`itr1==itr2`**: returns true if the iterators refer to the same location
- **`itr1!=itr2`**: returns true if the iterators refer to a different location
- Example:

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


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

Container Operations that Require Iterators

- **`iterator insert(iterator pos, const Object & x)`**: adds x into the list prior to the position given by the iterator; returns an iterator representing the position of the inserted item
- **`iterator erase(iterator pos)`**: removes the object at the position given by the iterator; returns the position of the element that followed pos.
- **`iterator erase(iterator start, iterator end)`**: removes all items beginning at position start up to but not including end.

  Example:

```
c.erase( c.begin(), c.end() )
```

Example

```
template <typename Container>
void removeEveryOtherItem( Container & lst )
{
    typename Container::iterator itr = lst.begin( );
    //C++11: auto itr = lst.begin( );

    while( itr != lst.end( ) )
    {
       itr = lst.erase( itr );
       if( itr != lst.end( ) )
          ++itr;
    }
}
```

=> T(N) = O(N) if list

T(N) = $O(N^2)$ if vector

const_iterators

- **`const_iterator`** returns a constant reference
- **`const_iterator`** cannot appear on the left-hand side of an assignment statement
- The compiler will force the use of **`const_iterator`** to traverse a constant collection

Implementation of vector

```
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 )  //C++11: use nullptr
             { 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;
       }

        //C++11: need to also implement a move constructor and a move assignment (for
       rhs that is an rvlaue, i.e., a temporary that is about to be destroyed)
```

Implementation of vector

```
void resize( int newSize )
{
     if( newSize > theCapacity )
         reserve( newSize * 2 );
     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;
}
```

Implementation of vector

```
Object & operator[]( int index )
      { return objects[ index ]; }
const Object & operator[]( int index ) const
      { return objects[ index ]; }

bool empty( ) const
      { return size( ) == 0; }
int size( ) const
      { return theSize; }
int capacity( ) const
      { return theCapacity; }

void push_back( const Object & x )
      {
      if( theSize == theCapacity )
         reserve( 2 * theCapacity + 1 );
      objects[ theSize++ ] = x;
      }

//C++11: need to also implement void push_back (Object && x)

void pop_back( )
      { theSize--; }

const Object & back ( ) const
      { return objects[ theSize - 1 ]; }
```

Implementation of vector

```
typedef Object * iterator;
typedef const Object * const_iterator;

iterator begin( )
     { return &objects[ 0 ]; }

const_iterator begin( ) const
     { return &objects[ 0 ]; }

iterator end( )
     { return &objects[ size( ) ]; }

const_iterator end( ) const
     { return &objects[ size( ) ]; }

static const int SPARE_CAPACITY = 16;

private:
     int theSize;
     int theCapacity;
     Object * objects;
};
```