Data Structures

vectors,lists,stacks and queues

Hikmat Farhat

June 5, 2018

Introduction

- "Linear" Containers in C++. A C++ container "contains" a set of objects. The objects can be of any type (the same).
- Implemented in the STL in C++
- But later we will implement our own version
- This will allow us to assess complexity of operations.
- A container of objects usually implements the following operations
  - create: create the container.
  - Add an element. The "position" where the element is added depends on the type of container.
  - erase: delete an item from a certain position or a range. Also depends on the type of container
  - empty: test wether the container is empty or not.
  - find: search for the existence of an element in the container.

- Since a container can store any type of elements we need to use templates to define them.

```
#include <iostream>
template <typename T>
class MemCell {
  T val;
   public:
  MemCell(T x){
      val=x;
  }
  T getVal(){return val;}
  void setVal(T x){val=x;}
};
int main(){
MemCell<int> m(10);
MemCell<std::string> mm("test");
m.setVal(23);
std::cout<<m.getVal()<<std::endl;
std::cout<<mm.getVal()<<std::endl;
}
```

STL vector

- One data structure provided by the Standard Template Library (STL) is the **vector** class.
- First we will use the STL vector class and see how it implements the vector ADT.
- A vector ADT is a suitable when
  - Elements are added/deleted only from the end of the list
  - Finding element at position k is used often and must be fast.
- It is fast in access elements at random positions
- More importantly: a vector ADT is not suitable when we need to add/remove the front element.

Iterators

- There are many times where we need to "iterate" through the elements of a list.
- We would like to do this regardless how the container is implemented.
- A convenient way of doing this is for the container to supply us with an **iterators**
- An iterator is simply a pointer to an element of the list
- An iterator supports increments methods and dereference operator to retrieve the value it points to.

Using Iterators to print all elements of vector

```
#include <iostream>
#include <vector>
#include <string>

int main(){
 std::vector<std::string> v;
 v.push_back("first string");
 v.push_back("second string");
 v.push_back("third string");

 std::vector<std::string>::iterator itr;

 for(itr=v.begin();itr!=v.end();itr++)
         std::cout<<*itr<<std::endl;

}
```

Using Iterators to insert and remove elements

```
#include <iostream>
#include <vector>
#include <string>
int main(){
    std::vector<std::string> v;
    v.push_back("first string");
    v.push_back("second string");
    v.push_back("third string");
    std::vector<std::string>::iterator itr;
    itr=v.begin();
    itr++;
    v.insert(itr,"between 1 & 2");
    itr=v.end();
    itr-=2;
    v.erase(itr);
    for(itr=v.begin();itr!=v.end();itr++)
        cout<<*itr<<endl;

    return 0;
}
```

STL list class

- A different implementation of the list ADT is the STL **list**
- Unlike the vector class which uses an array for internal storage
- the **list** is implemented as a linked list.
- Unlike **vector** it provides an efficient implementation of push_front and pop_front methods.
- Unlike **vector** it does NOT provide an efficient implementation of element at position k.

Using STL list

```
#include <iostream>
#include <list>

using namespace std;

int main (int argc, const char * argv[]){
   list<string> mylist;
    mylist.push_front("first element");
    mylist.push_back("second element");
    mylist.push_front("third element");

    list<string>::iterator itr;

for(itr=mylist.begin();itr!=mylist.end();itr++)
        cout<<*itr<<endl;

    return 0;
}
```

Need for copy constructors

```
#include <iostream>

class IntCell{
   int *val;
   public:
   IntCell(int x=0){
      val=new int(x);
   }
   int getVal(){return *val;}
   void setVal(int x){*val=x;}
};
int main(){
IntCell a(10);
IntCell b=a;
a.setVal(20);
std::cout<<a.getVal()<<std::endl;
std::cout<<b.getVal()<<std::endl;
}
```

Need for copy constructor

- The output for both *a* and *b* is 20. This is because the default copy constructor copies element by element.
- So instead of copying the value stored in *a* it copies the value of the pointer.
- We need to provide our own version of the copy constructor

```
IntCell(const IntCell & rhs){
val=new int(*rhs.val);
}
```

Why a copy constructor?

- A copy constructor is used by the compiler in the following cases
  - When an argument is passed by value, a copy of the argument should be made
  - when a function returns a local object (not a pointer or reference to it), an anonymous and temporary copy should be made to be returned to the caller.
  - when a object is initialized as: Type obj(initial_obj) then obj is made as a copy of initial_obj using the copy constructor.
- the compiler usually supplies a default copy constructor.
- As we have seen when there is dynamic memory allocation this default copy constructor does not work.

Vector interface

```
#ifndef list_vector_Vector_h
#define list_vector_Vector_h
template <typename Object>
class Vector
{
 private:
  int theSize;
  int theCapacity;
  Object * objects;

 public:
  explicit Vector(int initSize=0);
  Vector( const Vector & rhs );
  ~Vector( );
  const Vector & operator= ( const Vector & rhs );
  void resize( int newCapacity );
  Object & operator[]( int index );
  bool empty( ) const;
  int size( ) const;//how many elements?
  int capacity( ) const;//total capacity
  void push_back( const Object & x );
  void pop_back( );
  typedef Object * iterator;
  iterator begin( );
  iterator end( );

};
```

Adding an element to Vector

```
template <typename Object>
void Vector<Object>::push_back( const Object & x )
{
  if( theSize == theCapacity )
    resize( 2 * theCapacity );
  objects[ theSize++ ] = x;
}
```

- In the code above sometimes we need to call the expensive *resize* .

```
template <typename Object>
void Vector<Object>::resize( 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;
}
```

Vector Implementation

```
template <typename Object>
Vector<Object >::Vector(int initSize)
:theSize(initSize),theCapacity(initSize)
{ objects=new Object[theCapacity];}

template <typename Object>
Vector<Object >::Vector(const Vector<Object>& rhs):theSize(rhs.size()),
theCapacity(rhs.theCapacity)
{

  objects = new Object[ capacity( ) ];
  for( int k = 0; k < size( ); k++ )
    objects[ k ] = rhs.objects[ k ];

}

template <typename Object>
Vector<Object >::~Vector()
{delete[] objects;}

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

Vector Implementation

```
template <typename Object>
void Vector<Object>::resize( 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;
}

template <typename Object>
Object & Vector<Object>::operator[]( int index )
 { return objects[ index ]; }

template <typename Object>
bool Vector<Object>::empty( ) const
{ return size( ) == 0; }
```

- inserting elements in the vector is a costly operation because a whole portion of the array needs to be copied.
- the worst case happens when the insertion is done on the front.
- This is why the vector class supports inserts at the end only.
- Even in that case it becomes costly when we run out of space and we need to increase the storage
- check the method resize(int) in our implementation of vector.

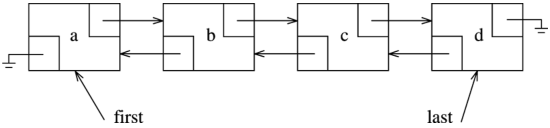
why a destructor?

- When an object goes out of scope it is destroyed.
- This is done by calling the destrcutor method.
- if no destrcutor method is specified the compiler uses a default one.
- If memory was allocating dynamically through a pointer, the default destructor destroys the pointer and NOT the memory that the pointer points to.
- therefore when memory is allocated dynamically it should be destroyed manually through the destructor.

Linked lists

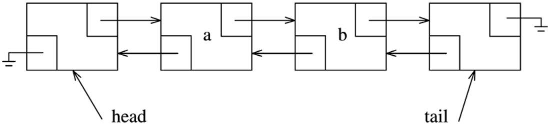
- We need to be able to change the size of the list dynamically. this is not possible with an array implementation.
- the implementation of insert and delete is very inefficient.
- An implementation that satisfies the above two condition is a **linked list** structure:
  - Elements do not need to be store consecutively.
  - But then we need to **link** the elements together.

- A linked list is a sequence of **nodes**
- Each node has two parts:
  - Data part: information stored in that particular node.
  - Next part: a link (pointer) to the next node.
  - We only need to have a pointer to the first node.
  - the next part of the last node is null.
  - In this case we use a **doubly** linked list where each node has a next and prev pointers.



Sentinel Node

- For convenience, we use empty nodes, called **sentinel**, for head and tail.
- The first element in the list is the node just after the head.
- The last element in the list is the node just before the tail.



Operations on Linked Lists

We would liked to have the following operations implemented on linked lists.

- create a list.
- test if the list is empty.
- display the list.
- search for an item in the list.
- delete an item from the list.
- insert an item into the list.

- Most operations will make use of an iterator
- In this case an iterator will be a node with extra operations
- Like itr++, itr–, itr=, itr!=, *itr
- All elements of the list are also accessed through iterators

```
template <typename Object>
struct Node{
    Object data;
    Node *prev;
    Node *next;
    Node( const Object & d=Object (), Node *p=NULL,
          Node *n=NULL)
    :data(d),prev(p),next(n){}
};
```

List interface

```
template <typename Object>
class List{

private:
int   theSize;
  Node<Object> *head;
  Node<Object> *tail;
  void init( ) { //... }

public:
  class iterator{
//code for iterator here
 };

  List(){//...}
  List(const List &rhs){//...}
  ~List() {}
  List & operator=(const List &rhs){//...    }
```

List interface CONT

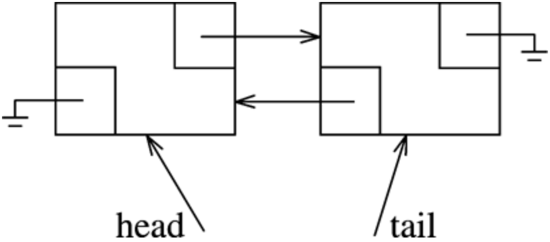
```
iterator begin(){ //...}
iterator end()  { //...}
int size() {//...}
bool empty() {//...}
void clear(){//...  }
void push_front( const Object & x ){//...}
void push_back(const  Object & x ){//...}
void pop_front( ){ //...}
void pop_back( ){//...}
iterator insert( iterator itr, const Object & x )  {  }
iterator erase( iterator itr ){    }
 };
```

Iterator Class

```
class iterator{
 protected:
   Node<Object> *current;
 public:
   iterator(){}
 iterator(Node<Object> *p):current(p){}
 Object & operator*()   {return current->data;}
 iterator & operator++(){
     current=current->next;
     return *this;
  }
   iterator & operator++(int in){
     current=current->next;
     return *this;
   }
   iterator & operator--(){
     current=current->prev;
     return *this;
   }
   iterator & operator--(int in){
     current=current->prev;
     return *this;
   }
   bool operator==(const  iterator & rhs) const
   {return current==rhs.current;}
  bool operator!=(const iterator &rhs) const
  {return !(current==rhs.current);}
  friend class List<Object>;
};
```

Empty list

- Since we always have sentinel nodes an empty list has two nodes: head and tail



beginning and end

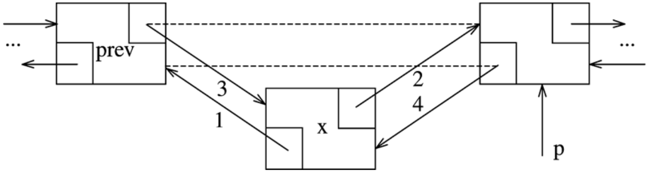
- The beginning and end are usually used differently
- for example

```
for(itr=list.begin();itr!=list.end();itr++)
```

- In the code above, begin() should return the first element (i.e the one after head)
- Whereas end should return tail thus

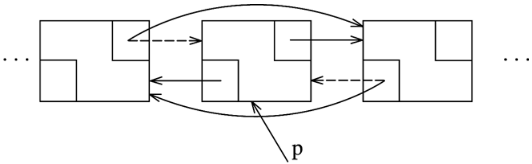
```
iterator begin(){
  return iterator(head->next);
}
iterator end(){
  return iterator(tail);
}
```

Inserting a Node




```
iterator insert( iterator itr , const Object & x )
{
  Node *p=itr.current;
  theSize++;
  Node *newNode=new Node(x,p->prev,p);
  p->prev->next=newNode;
  p->prev=newNode;
  return iterator(newNode);
}
```

Deleting a node




```
iterator erase( iterator itr ){
   Node *p=itr.current;
   iterator ret(p->next);
   p->prev->next=p->next;
   p->next->prev=p->prev;
   delete p;
   theSize--;
   return ret;
 }
```

List Destructor

- Every time we add a node to the list we allocate additional memory.
- When the list is out of scope and need to be destroyed, dynamically allocated memory is not destroyed automatically.
- Therefore we need to provide an explicit destructor for the dynamically allocated memory.
- Similarly we need to provide a copy constructor and define an "assignment" operator.

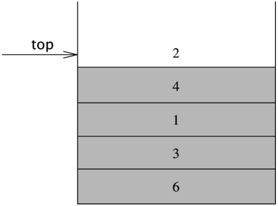
```
~List (){
  clear();
  delete head;
  delete tail;
}
void clear(){
   while(!empty())
     erase(begin());
}
```

Difference between vector and list

- Vector
  - Insertion and deletion are $\Theta(n)$
  - Direct access is $\Theta(1)$
- Linked list
  - Insertion and deletion are $\Theta(1)$
  - Direct access is $\Theta(n)$

Stack ADT

- A stack is a list of elements where only the **top** element is accessible
- Operations are
  - **push** to put a new element at the top
  - **pop** to remove the top element



Stack implementation

- We can use linked list but since insertion and deletion is done only at the top it is better to use an array
- Since only the top of the stack is accessible, insertion and deletion is done efficiently
- We need the following operations top(),push(),pop()
- The stack has :
  - capacity
  - top of stack
  - array of objects

Stack Interface

```
template <typename Object>
class Stack
{
 private:
  int topOfStack;
  int theCapacity;
  Object * objects;
  void reserve( int newCapacity );
 public:
Stack(int capacity=16):theCapacity(capacity)
    {
      topOfStack=-1;
      objects=new Object[theCapacity];
    }
```

Stack Interface cont.

```
int capacity(){return theCapacity;}
Stack( const Stack & rhs ){
 if(this != &rhs){
  theCapacity=rhs.theCapacity;
  topOfStack=rhs.topOfStack;
  objects=new Object[theCapacity];
 for(int i=0;i<theCapacity;i++)
   objects[i]=rhs.objects[i];
 }
}

int size( ) const{
   return topOfStack+1;
}
~Stack( ){
   delete[] objects;
}
Stack & operator= ( const Stack & rhs );//defined later
```

Stack Interace cont.

```
void push(const Object & x){
    if( topOfStack== theCapacity-1 )
      reserve( 2 * theCapacity + 1 );
    objects[++topOfStack]=x;
}
void pop(){
  if(!empty())
    topOfStack--;
}
Object top(){
    return objects[topOfStack];
  }
};
```

Assignment Operator

```
template <typename Object>
Stack<Object> & Stack<Object >::operator=
 ( const Stack<Object>& rhs )
   {
        if( this != &rhs )
        {
            delete [ ] objects;
            topOfStack = rhs.size( )-1;
            theCapacity = rhs.theCapacity;

            objects = new Object[ theCapacity ];
            for( int k = 0; k < size( ); k++ )
                objects[ k ] = rhs.objects[ k ];
        }
        return *this;
    }
```

Stack Application: Postfix Calculator

- "regular" expressions are called infix expressions:

$$17 + 3 * 5$$

- is interpreted as:

$$17 + (3 * 5)$$

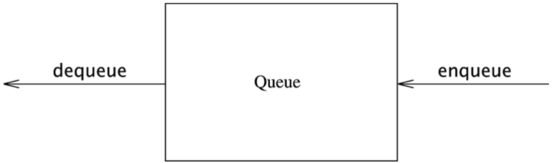
- because $*$ has higher precedence than $+$.
- Postfix expressions are easier to evaluate because we don't need to remember precedence rules. The above in postfix notation is

$$3\ 5 * 17 +$$

- A postfix calculator can be implemented using a stack as follows
  1. If a number is read then it is **pushed** on the stack.
  2. when an operator is read then
     1. the appropriate number of arguments (usually two) are **poped** from the stack.
     2. the operator is applied to the arguments
     3. The results is **pushed** back onto the stack.
- Example, evaluate 6 5 2 3 + 8 * + 3 + *

Queue ADT

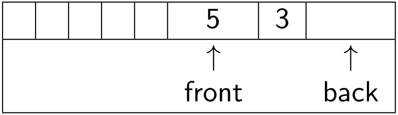
- The basic operations on a Queue are
  1. Enqueue to add an element to the **end** of a list
  2. Dequeue to return (and remove) the **front** element of a list
- This why sometimes it is called First in First Out (FIFO).



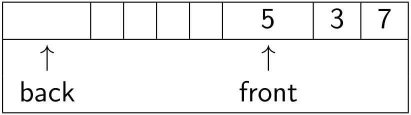
Array Implementation of Queue ADT

- A queue can be implemented using an array by maintaining two values
  - front that points to the first element
  - back that points to end of the queue (last+1).

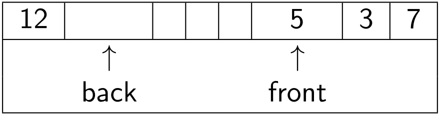
Initially



After enqueue(7)



After enqueue(12)



After dequeue()