VE280 2020FA Final Review Part2

L19: Linked List

Expandable arrays are only one way to implement container that can grow and shrink at runtime. To enlarge a list implemented by linked list, you can simply add a node at the end of the linked list.

Review what you have implemented for *p5* thoroughly.

Single-Ended & Double-Ended

Linked lists could be either single-ended or double-ended, depending on the the number of node pointers in the container.

In a single-ended list, we only need to maintain `first`.

```
class IntList {
    node *first;
    //...
};
```

In a double-ended list, we introduce `last`.

```
class IntList {
    node *first;
    node *last;
    //...
};
```

Especially, when handling a singly ended list, you need to be concerned about the **boundary situation** where

- size = 0: `first = nullptr`

In a double-ended list, the `last` makes it slightly more complicated:

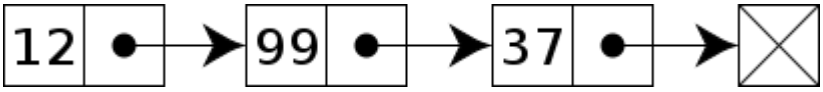
- size = 0: `first = last = nullptr`
- size = 1: `first = last`.

Singly-Linked & Doubly-Linked

Linked lists could be either single linked or doubly linked, depending on the the number of directional pointers in `node`.

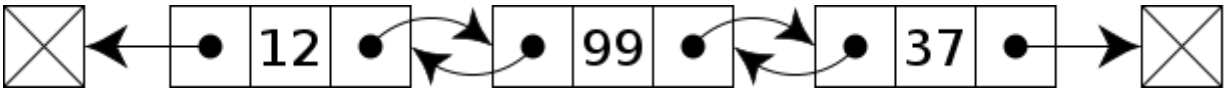
In a singly linked list, we only need a `next`.

```
struct node {
    node *next;
    int value;
};
```



In a doubly linked list, we need also a `prev`.

```
struct node {
    node *next;
    node *prev;
    int value;
};
```



L20: Template and Container

Template Function

A single function can also be templated to implement generic algorithms.

Note:

- **Using `class` and `typename` are both fine. `typename` is introduced in C++11, they are basically the same.**
- `template <...>` must be written just before the header of the function/class, because they are actually a single declaration statement: `template <typename T> typename T add(const T &num1, const T &num2)`. Usually they are written in two lines for clarity.

```
template <typename T>
typename T add(const T &num1, const T &num2) {
    // For int, double... This will be sum of numerical value
    // For other objects, you can overload the "+" operator
    return num1 + num2;
}
```

No need to specify type when using:

```
double a = 2.0, b = 3.0;
add(1, 2);
double c = add(a, b);
```

Template Class

```
template <class T1, class T2, ...>
class ClassName {
// Defination of class
}
```

- Usually used to write containers (A special kind of class that only holds data, but not do any specific modification to them)
- The typenames will be replaced with specific types when compiling (similar to Macro during preprocess):
  - Therefore, more types you use, the compiling time will be longer
  - More types you use, the size of final program will be larger
- Definition and declaration are usually written in one header file:
  - C++ will compile each .cpp file into an object file (.o). However, it's meaningless to compile a template without specific types into an object file (similar logic as an abstract class will not have instances).

Container of Pointers

Modification of Declaration

Because copying objects (when passed as parameters or assigning) waste a lot of time and memory, we would like to store the addresses only. However, we will not write like this:

```
template <class T>
class List {
  public:
    ...
    void insert(T v);
    T    remove();
  private:
    struct node {
        node *next;
        T     o;
    };
    ....
};
```

This still allows users of this container to declare a container that stores values rather than pointers by writing: `List<BigThing> ls;` In order to eliminate this possibility, we would rather write:

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T    *remove();
  private:
    struct node {
        node *next;
        T    *o;
    };
    ....
};
```

One Invariant & Three Rules

There are some basic rules to ensure there will be no memory issues:

- **At-most-once invariant**: any object can be linked to at most one container at any time through pointer.
- **Existence**: An object must be **dynamically allocated** before a pointer to it is inserted.
- **Ownership**: Once a pointer to an object is inserted, that object becomes the property of the container. It can only be modified through the methods of the container. When an object exists in memory, any object or function that has a pointer or reference to the object could potentially modify it (**unless the pointer or reference is declared const**).
- **Conservation**: When a pointer is removed from a container, either the pointer must be inserted into **some** container, or its referent must be **deleted**.

Analogy: Let's compare object to money, compare container to banks.

- **Invariant**: A sum of money can only be held by someone, or some bank, not both.
- **Existence**: You must have some money before you put them into bank. The bank itself cannot generate money.
- **Ownership**: Once your money is put into the bank, you can only go the bank counter or the ATM to manage them.
- **Conservation**: When you take your money out, you can only put it into other banks, or manage by yourself. (Of course you can throw it away if you are rich)

Example:

As an example, assume you have the following class definitions:

```
class Container {
  public:
    // construct a container instance by making it point to an item
    Container(Item *pItem) : m_pItem(pItem) { }
  private:
    Item *m_pItem;
};

class Item {
  public:
    change() {
      // do something to change the item
    }
  private:
    // some private data
};
```

```
Item *pItem = new Item();  // the function containing this code obviously
has a pointer
                           // to the newly created item
Container c(pItem);        // let the container c own this item

pItem->change();         // attempt to modify an object owned by c
```

```
class Container {
  public:
    removeItem() {
      delete m_pItem;  // deallocate the item instance owned by this
container
                       // the item is no longer owned because it no longer
exists
    }
    giveItemTo(const Container& other) {
      other.m_pItem = m_pItem;  // let the other container own this item
instead
      m_pItem = NULL;           // the item is no longer owned by this
container because the pointer is NULL
    }
};
```

```
Item *pItem = new Item();  // construct a new Item
Container c1(pItem);       // create a new container pointing to this item
Container c2(pItem);       // WRONG: create another container pointing to
the same item
```

```
Item *pItem = NULL;
Container c1(pItem);  // pItem is not dynamically allocated; it is NULL

Item item;
Container c2(&item);  // &item does not point to a dynamically allocated
heap
                      // object; the object is on stack;
```

Some student's P5: violate which law?

```
int *num1 = new int(*numList->removeFront());
int *num2 = new int(*numList->removeFront());
numList->insertFront(num1);
numList->insertFront(num2);
```

Destroy a templated Container

According to our rules, a container must be responsible for the objects it is holding. If a container is destroyed before the objects in it is deleted, there will be memory leak. Therefore, we should rewrite:

- **The destructor**: Destroys an existing instance.
- **The assignment operator**: Destroys an existing instance before copying the contents of another instance. (What will happen if self-assignment?)

Lecture 22: Linear List; Stack

Linear List

A collection of zero or more integers; duplicates possible. It supports insertion and removal by position.

Insertion

```
void insert(int i, int v) // if 0 <= i <= N (N is the size of the list),
//insert v at position i; otherwise, throws BoundsError exception.
Example: L1 = (1, 2, 3)
L1.insert(0, 5) = (5, 1, 2, 3);
L1.insert(1, 4) = (1, 4, 2, 3);
L1.insert(3, 6) = (1, 2, 3, 6);
L1.insert(4, 0) throws BoundsError
```

Removal

```
void remove(int i) // if 0 <= i < N (N is the size of the list), remove
the i-th
element; otherwise, throws BoundsError exception.
Example: L2 = (1, 2, 3)
L2.remove(0) = (2, 3);
L2.remove(1) = (1, 3);
L2.remove(2) = (1, 2);
L2.remove(3) throws BoundsError
```

Stack

Property: **last in, first out (LIFO)**

- `size()` : number of elements in the stack.
- `isEmpty()` : check if stack has no elements.
- `push(Object o)` : add object o to the top of the stack.
- `pop()` : remove the top object if stack is not empty; otherwise, throw stackEmpty .
- `Object &top()` : returns a reference to the top element.

Implementation Comparison

1. Using Arrays Need an int size to record the size of the stack.

- `size()` : return size;
- `isEmpty()` : return (size == 0);
- `push(Object o)` : add object o to the top of the stack and increment size . Allocate more space if necessary.
- `pop()` : If isEmpty() , throw stackEmpty ; otherwise, decrement size . Object &top() : returns a reference to the top element Array[size-1].

2. Using Linked Lists

- `size()` : LinkedList::size();
- `isEmpty()` : LinkedList::isEmpty();
- `push(Object o)` : insert object at the beginning LinkedList::insertFirst(Object o); .
- `pop()` : remove the first node LinkedList::removeFirst(); Object &top() : returns a reference to the object stored in the first node

Lecture 23: Queue

- `size()`: number of elements in the queue.
- `isEmpty()`: check if queue has no elements.
- `enqueue(Object o)`: add object o to the rear of the queue.
- `dequeue()`: remove the front object of the queue if not empty; otherwise, throw queueEmpty.
- `Object &front()`: return a reference to the front element of the queue.
- `Object &rear()`: return a reference to the rear element of the queue.

Queues Using Linked Lists

How to implement the mothods using linked lists:

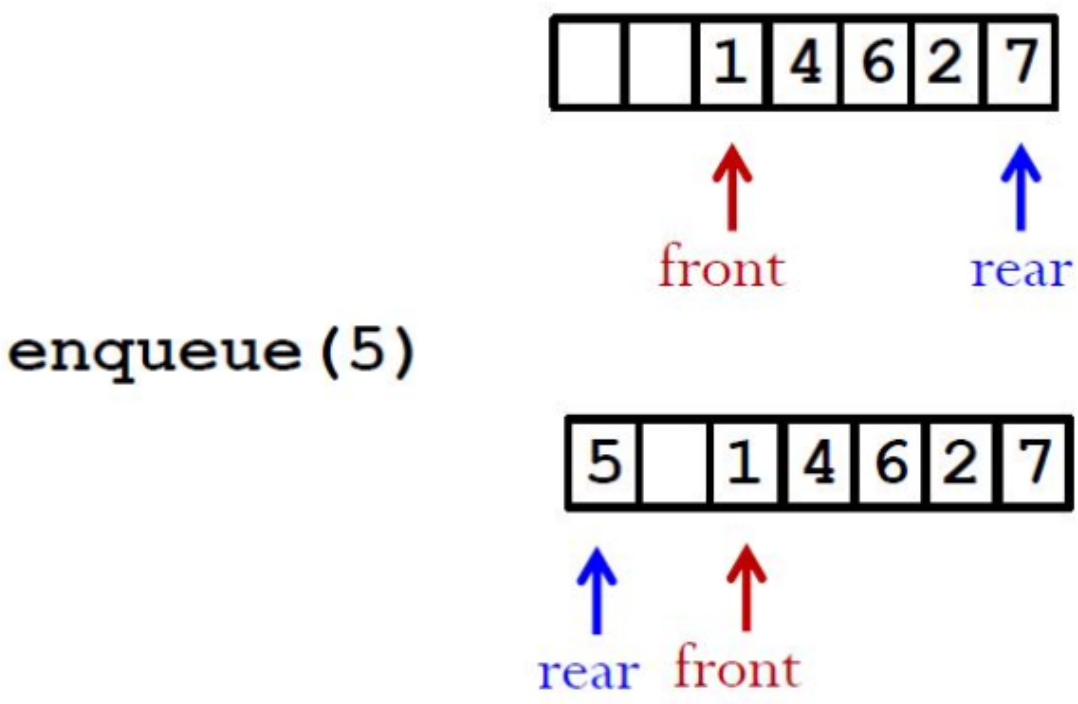
- `enqueue(Object o)`: add a node to the end of the linked list.
- `dequeue()`: remove a node from the head of the linked list.
- `size()`: can iterate through the linked list and count the number of nodes.
- `isEmpty()`: check if the pointer to the linked list is NULL.
- `Object &front()`: returns a reference to the node at the head of the linked list.
- `Object &rear()`: returns a reference to the node at the end of the linked list.

Queues Using Arrays

Let the elements "drift" within the array.

Maintain two integers to indicate the front and the rear of the queue (advance front when dequeuing; advance rear when inserting).

Use a circular array (more space efficient):



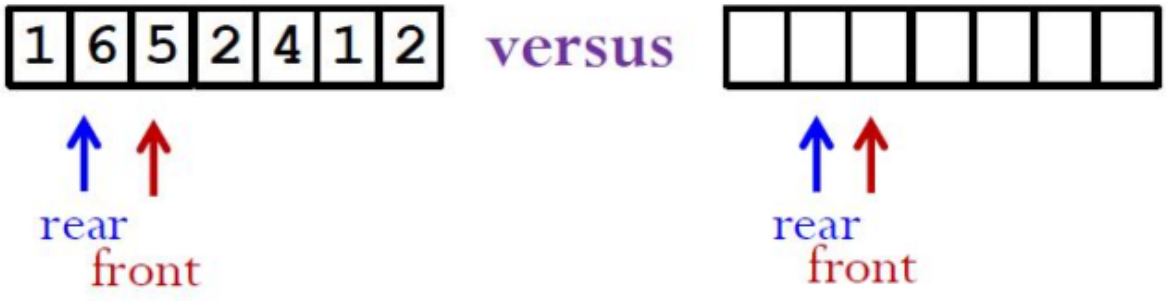
When inserting a new element, advance rear circurlarly; when popping out an element,

advance front circularly.

Can be realized by

```
front = (front+ 1 ) % MAXSIZE;
rear = (rear+ 1 ) % MAXSIZE;
```

Solve the problem of distinguishing an empty queue and full queue:



Maintain a `flag` indicating empty or full, or a `count` on the number of elements in the queue.

We can see that using array can be more complicated than linked list, and the size can be restricted by `MAXSIZE` of array. However, it can be easier to access the elements by index. For example, the index of the second element can be `front+1` (if not exceeding the capacity of array). Still, accessing elements by index is not very useful in queues.

Deque

- Not a proper English word, pronounced as “deck”.
- Means double-ended queue
- Property: Items can be inserted and removed from both ends of the list.
- Methods:
    - `push_front(Object o)`
    - `push_back(Object o)`
    - `pop_front()`
    - `pop_back()`

The implementation can be more complicated than queue. Only use it if **inserting and removing from both ends are truly necessary**.

Lecture 24: STL

Three kinds of containers:

- Sequential Containers : let the programmer control the order in which the elements are stored and accessed. The order doesn't depend on the values of the elements.
- Associative Containers : store elements based on their values.
- Container Adaptors : take an existing container type and make it act like a different type

Sequential Containers:

1. vector: based on arrays.

- fast random access
- fast insert/delete at the back
- inserting / deleting ar other position is slow

2. deque (double-ended queue): based on arrays.
   - fast random access (operator[])
   - fast insert/delete at front or back
3. list: based on doubly-linked lists.
   - only bidirectional sequential access (no operator[])
   - fast insert/delete at any point in the list

I. Vector

```
#include <vector>
using namespace std;
```

Initialization

```
vector<T> v1; // empty vector v
vector<T> v2(v1); // copy constructor
vector<T> v3(n,t); // construct v3 that has n elements with value t
```

Size

- v.size() returns a value of size_type corresponding to the vector type.

```
vector<int>::size_type
```

- a companion type of vector (why: to make the type machine-independent)
- essentially an unsigned type: you can convert it into unsigned int but not int

```
unsigned int s = v.size();
```

check whether empty

```
v.empty()
```

Add/Remove

- v.push_back(t): add element t to the end of v
    - container elements are copies: no relationship between the element in the container and the value from which it was copied
- v.pop_back(): remove the last element in v. v must be non-empty

1. Subscripting Vector

```
vector<int>::size_type ix;
for (ix = 0 ; ix!= v.size(); ++ix) {
v[ix] = 0 ;
}
```

Subscripting doesn't add elements

```
vector<int> ivec; // empty vector
for(vector<int>::size_type ix= 0 ; ix!= 10 ; ++ix)
ivec[ix] = ix; // Error!
```

```
v1 = v2;//replace elements in v1 by a copy of elements in v
v.clear();//makes vector v empty
v.front();//Returns a reference to the first element in v.
//v must be non-empty!
```

Iterator

All of the library containers define iterator types, but only a few of them support subscripting.

- Declaration:

```
vector<int>::iterator it;
```

- `v.begin()` returns an iterator pointing to the first element of vector
- `v.end()` returns an iterator positioning to **one-past-the-end** of the vector
    - usually used to indicate when we have processed all the elements in the vector
    - If the vector is empty, the iterator returned by begin is the same as the iterator returned by end, e.g.
- Operations:

- dereference: can read/write through *iter (cannot dereference the iterator returned by end())
- ++iter, iter++: next item
- --iter, iter-- go back to the previous item
- iter == iter1 and iter != iter1: check whether two iterators point to the same data item

What happens when ivec is empty?

Why using iterator instead of subscript?

II. Deque

```
#include <deque>
```

Similarities with vector:

Initialization:

- `deque<T> d; deque<T> d(d1);`
- `deque<T> d(n,t)`: create d with n elements, each with value t.
- `deque<T> d(b,e)`: create d with a copy of the elements from the range denoted by iterators b and e.
- Difference with vector:
  - `d.push_front(t)`
  - `d.pop_front(t)`

III. List

```
#include <list>
```

only bidirectonal sequential access

3. Similarities with vector:

Initialization:

4. Difference with vectors

- do not support subscripting
- not suppoer iterator arithmetic, i.e. cannot do it+3, you can only use ++/--
- no relational operation <, <=, >, >=, you can only use == and != to compare
- `l.push_front(t)`
- `l.pop_front(t)`

Choose appropriate containers

1. Use vector, unless have other good reasons
2. require random access: vector or deque
3. insert or delete elements in the middle: list
4. insert or delete elements at the front and the back: deque
5. others: check the predominant operations