stack

- stack是一种container adapter(容器适配器)，以某种既有容器作为底层结构，将其接口改变，使之符合先进后出(Frist In Last Out，FILO)的特点，默认情况下使用deque(双端队列)作为底层容器。

stack的使用

- stack()
  - 构造空的栈
- bool empty()
  - 如当前堆栈为空，返回 true 否则返回false
- size_t size()
  - 返回当前堆栈中的元素数目
- value_type& top()
  - 返回对栈顶元素的引用
- void push(const value_type& val)
  - 将 val值压栈，使其成为栈顶的第一个元素
- void pop()
  - 移除栈中最顶层元素

stack的实现

```
#pragma once
#include<deque>

namespace bin
{
    template<class T,class Container=deque<T>>
    class stack
    {
    public:

        void push(const T& data)
        {
            _con.push_back(data);
        }

        void pop()
        {
            _con.pop_back();
        }

        size_t size()
        {
            return _con.size();
        }

        bool empty()
```

```
        {
            return _con.empty();
        }

        T& top()
        {
            return _con.back();
        }
    private:
        Container _con;
    };
}
```

queue

- queue是一种容器适配器，以某种既有容器作为底层结构，将其接口改变，使之符合先进后出(Frist In First Out，FIFO)的特点，默认情况下使用deque(双端队列)作为底层容器。

queue的使用

- queue()
  - 构造空队列
- bool empty()
  - 如果队列为空返回真(true)，否则返回假(false)
- size_t size()
  - 返回队列中元素的个数
- value_type& front()
  - 返回队列第一个元素的引用
- value_type& back()
  - 返回一个引用，指向队列的最后一个元素
- void push(const value_type& data)
  - 往队列中加入一个元素
- void pop()
  - 删除队列的一个元素

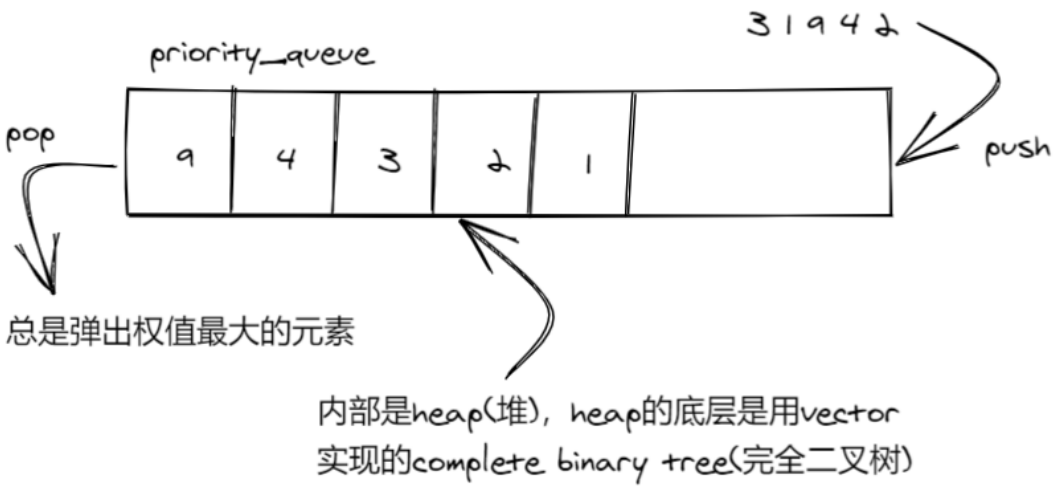
queue的实现

```
#pragma once
#include<deque>

namespace bin
{
    template<class T, class Container=deque<T>>
    class queue
    {
    public:

        //
        void push(const T& data)
        {
```

```
            _con.push_back(data);
        }

        void pop()
        {
            _con.pop_front();
        }

        size_t size()
        {
            return _con.size();
        }

        bool empty()
        {
            return _con.empty();
        }

        T& front()
        {
            return _con.front();
        }

        T& back()
        {
            return _con.back();
        }
    private:
        Container _con;
    };
}
```

priority_queue

- priority_queue是拥有权值队列的queue，内部的元素是自动按照权值排列，权值最高者排在最前面，缺省情况下priority_queue利用一个max_heap(大堆)——用vector表示的compele binary tree(完全二叉树)，max_heap可满足priority_queue需要的按照权值高低排序的特性。



priority_queue的使用

- priority_queue()/priority_queue(frist,last)
  - 构造一个空的优先级队列
- bool empty()
  - 如果优先队列为空返回真(true)，否则返回假(false)
- value_type& top()
  - 返回一个引用，指向优先队列中有最高优先级的元素
- void push(const value_type& data)
  - 添加一个元素到优先队列中，值为data
- void pop()
  - 删除优先队列中的第一个元素

priority_queue的实现

```
#pragma once
#include<vector>
#include<iostream>
using namespace std;

namespace bin
{
    //仿函数 函数对象
    template<class T>
    struct less
    {
        bool operator()(const T& x1, const T& x2)
        {
            return x1 < x2;
        }
    };

    template<class T>
    struct greater
    {
        bool operator()(const T& x1, const T& x2)
        {
            return x1 > x2;
        }
    };

    //优先队列，默认是大堆
    template<class T,class Container=vector<T>,class Compare=less<T>>
    class priority_queue
    {
    public:
        //push相当于堆算法的向上调整
        void push(const T& data)
        {
            _con.push_back(data);
            AdjustUp(_con.size() - 1);
        }

        void pop()
        {
            swap(_con[0], _con[_con.size() - 1]);
```

```
            _con.pop_back();
            AdjustDown(0);
        }

        T& top()
        {
            return _con[0];
        }

        size_t size()
        {
            return _con.size();
        }

        bool empty()
        {
            return _con.empty();
        }
    private:
        void AdjustUp(int child)
        {
            Compare com;
            int parent = (child - 1) / 2;
            while (child > 0)
            {
                //if (_con[child] > _con[parent])
                if (com(_con[parent], _con[child]))
                {
                    swap(_con[parent], _con[child]);
                    child = parent;
                    parent = (child - 1) / 2;
                }
                else
                {
                    break;
                }
            }
        }
        void AdjustDown(int root)
        {
            Compare com;
            int parent = root;
            int child = parent * 2 + 1;
            while (child < _con.size())
            {
                //选出左右孩子中大的那一个
                //if (child + 1 < _com.size() && _con[child + 1] < _con[child])
                if (child + 1 < _con.size() && com(_con[child], _con[child +
1]))
                {
                    ++child;
                }
                //if (_con[child] < _con[parent])
                if (com(_con[parent], _con[child]))
                {
                    swap(_con[child], _con[parent]);
                    parent = child;
                    child = parent * 2 + 1;
```

```
                }
                else
                {
                    break;
                }
            }
        }
        Container _con;
    };
}
```