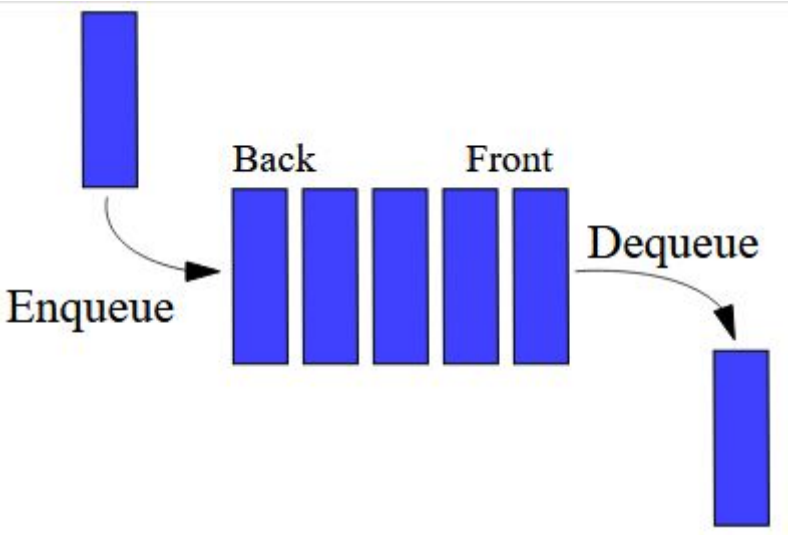
Universidad de Costa Rica
CI-0113: Programación 2
Laboratorio #5

Objetivo: Familiarizar al estudiante con el concepto y uso de listas en C/C++.

Enunciado

Resuelva los siguientes problemas. Escriba un código en C/C++ que resuelva los siguientes problemas:

1. La cola (queue) es una estructura de datos abstracta muy similar a las listas con la diferencia esencial de que en una cola los datos se ingresan a la lista (push/enqueue) por el último elemento (back), pero se sacan de la lista (pop/dequeue) por el frente (front). Este comportamiento se ve representado en la siguiente imagen:



Esta estructura tiene similitud con las listas en que puede ser implementado de las mismas maneras que una lista convencional: usando nodos simplemente enlazados, doblemente enlazados o arreglos.

Una variación de la estructura cola es la cola de prioridad (priority queue), en la cual los elementos no solo se ordenan en orden de llegada primero-en-entrar primero-en-salir (FIFO) sino que cada dato ingresado posee un valor de prioridad: la idea consiste en que el elemento con la mayor prioridad sea el primero en salir; y en caso de empate de prioridades entonces el primero en entrar es el primero en salir. Por lo tanto los datos no siempre se ingresan al final de la cola, sino que al ser ingresados deben buscar su ubicación correcta dentro de la fila, según su prioridad.

   a. Implemente su propia clase cola de prioridad que permita almacenar datos de tipo const char* (cadenas de hileras constantes). La prioridad será dada por un número entero, en el cual el número menor posee mayor prioridad. La información debe almacenarse utilizando nodos simple o doblemente enlazados.

b. Su clase debe poseer el método *push* que recibe por parámetro un puntero a una cadena constante de caracteres y un valor de prioridad. El objeto se debe almacenar en su ubicación correcta.
c. Su clase debe poseer el método *pop* que retorna la cadena constante de caracteres siguiente y la remueve de la cola.
d. Su clase debe poseer el método *size* que retorna la cantidad de elementos actualmente en la cola.
e. Implemente un código main que permita probar su código:

```
cola.push("Pedro",3);
cola.push("Majo", 2);
cola.push("Nacho",3);
cola.push("Fer",7);
```

Produciría la cola:
Majo -> Pedro -> Nacho -> Fer

Y Majo sería el primer elemento en salir de la cola al llamar pop();

2. (Opcional +10 puntos de honor) Al trabajar con estructuras de datos que podrían llegar a ser recorridas por los programadores puede resultar de utilidad crear un objeto iterador que permita recorrer los elementos del objeto de manera ordenada. Esto porque a veces realizar accesos de tipo *get* puede resultar muy lento para ciertas estructuras, en especial si se quiere acceder a todos los elementos de la lista de manera ordenada.
   a. Para implementar este tipo de objetos se crea una clase anidada Iterator, de manera que para referirnos a ella usamos el nombre `ClasePadre::Iterator`
   b. La clase iterador usualmente posee un constructor que recibe una referencia a un objeto de la clase de la cual es iterador (`ClasePadre* obj`) y con ella inicializa su posición en el primer elemento de la estructura.
   c. La clase iterador puede poseer un constructor vacío que genera un objeto iterador que se usa para representar el caso "final" del iterador (no el último ítem de la lista, sino un ítem inexistente)
   d. La clase iterador posee una sobrecarga en su operador ++ prefijo (`Iterator &operator++()`) que hace que el iterador actual avance al siguiente elemento.
   e. La clase iterador posee una sobrecarga en su operador != (`bool operator!=(const Iterator &it) const`) que permite comparar dos elementos iterador y saber si se encuentran en diferentes elementos de la estructura.
   f. La clase iterador posee una sobrecarga del operador * (`const data operator*()`) que permite obtener el dato en que se encuentra actualmente el iterador.
   g. La clase padre posee un método begin() que retorna un objeto iterador en el punto inicial de la estructura:

```
Iterator begin(){ return Iterator(this); }
```

h. La clase padre posee un método end() que retorna un objeto iterador que es un caso “final” del iterador:

```
Iterator end(){ return Iterator(); }
```

i. De esta manera, al querer iterar sobre la estructura podemos utilizar un ciclo for de la siguiente manera:

```
for(Parent::Iterator it = obj->begin(); it != obj->end();
++it){
	cout << *it << endl;
}
```

j. Implemente una clase Iterator dentro de la cola de prioridad que permita recorrer la estructura de manera secuencial.