deque<T>

Practica5

Generado por Doxygen 1.7.5

Lunes, 2 de Diciembre de 2013 15:09:34

Índice general

Capítulo 1

DEQUE

Versión

v0

Autor

Juan F. Huete

1.1. Introducción

En esta práctica nuestro objetivo es doble: por un lado implementar un nuevo tipo de dato, el DEQUE que representa una estructura de datos lineal, mientras que por otro hacer uso de dos de los tipos lineales que hemos visto, la lista (list) y el vector. Siguiendo este segundo objetivo, la representación interna del deque estará basada en estos tipos (aunque esto hace que algunas operaciones no sean del todo eficientes).

1.2. ¿Qué es un deque?

Un deque (Double-Ended-QUEue) un contenedor secuencial que permite el acceso aleatorio a los elementos (mediante valores enteros- índice), inserción y borrado eficiente (en tiempo constante, O(1)) si se realiza al final y comienzo del deque. Sin embargo, la inserción y el borrado son de orden lineal si se realizan en posiciones intermedias del deque. Es posible iterar sobre todos los elementos, desde el primero al último de la secuencia.

Ambos, vectores y deques, proporcionan una interfaz muy similar y pueden utilizarse para resolver los mismos problemas, pero internamente son muy diferentes.

Mientras que los vectores almacenan la información de forma contigua, teniéndose que re-dimensionar cuando crece su tamaño, los elementos en un deque se almacenan en diferentes bloques de memoria, manteniendo el deque la información necesaria para permitir el acceso directo a cualquiera de sus elementos en orden constante.

Normalmente, estos contenedores suelen ser implementados bajo la forma de matrices bidimensionales (como hemos dicho, nosotros utilizaremos otra alternativa). Sin embargo, están dotados de una interfaz que permite considerar los elementos secuencialmente.

Por tanto, el deque tiene una representación interna mas compleja que el vector, lo que le permite crecer de forma más eficiente bajo ciertas circunstancias, particularmente cuando se consideran secuencias muy grandes de elementos, donde para el vector el realojar todos los elementos tiene un elevado coste.

El número de elementos en el deque puede variar dinámicamente; la gestión de la memoria es automática.

1.3. Generar la Documentación.

Al igual que en la práctica anterior la documentación se entrega mediante un fichero pdf, así como mediante un fichero tgz que contiene todos los fuentes junto a los archivos necesarios para generar la documentación (en latex y html). Para generar los ficheros html con la documentación de la misma es suficiente con ejecutar desde la línea de comando

```
doxygen dox_deque
```

Como resultado nos genera dos directorios, uno con la documentación en html y el otro con la documentación en latex. Para generar la documentación en pdf podemos ejecutar

```
cd latex
make
```

Al hacer make se ejecuta una llamada al programa latex (si lo tenemos instalado) que como salida nos genera el fichero refman.pdf

Se entrega el fichero de especificación del deque.

- deque.h Se os pide implementar los distintos métodos así como el código necesario para demostrar el correcto funcionamiento del mismo.

Pasamos a detallar cada una de las partes de la práctica.

1.4. Representación del deque

Como se ha comentado, a la hora de almacenar los datos en un deque estos se distribuyen en bloques de memoria de un tamaño fijo. En nuestro caso, consideraremos que cada bloque podrá almacenar hasta un máximo de TBQ elementos, por ejemplo 10 (esta constante estará definida en la parte privada del deque). Obviamente, cuando un deque tiene más de TBQ elementos, necesitará ocupar más de un bloque.

Así, por ejemplo, si un deque tuviese que almacenar 26 caracteres, de la A a la - Z en este orden, los elementos se tendrán que distribuidos en al menos 3 bloques (posiblemente 4). Obviamente, el usuario (consumidor del tipo) ve los elementos como si realmente estuviesen almacenados de forma consecutiva, esto es, si tenemos un deque<char> dq, dq[0] tendrá la A, dq[1] la B, ..., y dq[25] contendrá la Z. En cualquier caso, internamente los elementos se distrubuyen entre los bloques del deque, en el siguiente ejemplo mostramos tres posibles alternativas.

```
ALT1
bq 1:  A B C D E F G H I J
bq 2:  K L M N O P Q R S T
bq 3:  U V W X Y Z # # # #

ALT2
bq 1:  # # # A B C D E F G
bq 2:  H I J K L M N O P Q
bq 3:  R S T  U V W X Y Z #

ALT3
bq 1:  # # # # # # # # A B
bq 2:  C D E F G H I J  K L
bq 3:  M N O P Q R S T U V
bq 4:  W X Y Z  # # # # # #
```

1.4.1. Inserción al Final

La principal ventaja que tiene esta estructura es que permite insertar elementos de forma eficiente por el inicio y el fin. Así, por ejemplo, si quisieramos insertar los números del 0 al 4 (en este orden) al final del deque ocurren dos situaciones: para la alternativas ALT1 y ALT2 será necesario el solicitar al sistema operativo un nuevo bloque en el que se podrán insertar los elementos. Sin embargo, para la alternativa ALT3 no será necesario hacer esta petición, quedando como se ilustra . En cualquier caso, la inserción al final implica situar el elemento al final del último bloque, salvo en caso en que este la última posición del bloque estuviese ocupada, donde sería necesario realizar la petición de un nuevo bloque.

```
ALT1
bq 1:  A B C D E F G H I J
bq 2:  K L M N O P Q R S T
bq 3:  U V W X Y Z 0 1 2 3
bq4:   4 # # # # # # # # #

ALT2
bq 1:  # # # A B C D E F G
bq 2:  H I J K L M N O P Q
bq 3:  R S T  U V W X Y Z 0
bq4:   1 2 3 4 # # # # # #

ALT3
bq 1:  # # # # # # # # A B
bq 2:  C D E F G H I J  K L
bq 3:  M N O P Q R S T U V
bq 4:  W X Y Z 0 1 2 3 4 #
```

1.4.2. Inserción al Inicio

De igual forma, si nuestro objetivo es insertar el número 5 al principio, para la alternativas ALT2 y ALT3 no hay problema, lo podemos ubicar justo delante de la A. Sin embargo, para la alternativa ALT1, el bloque bq1 está lleno, y por tanto debemos de solicitar un nuevo bloque al sistema operativo, que se ubicará antes del bloque primero. En las figuras mostramos cómo quedarían la estructuras al insertar los números del 5 al 9 al principio del deque (en este orden). En este caso, la inserción al inicio implica situar el elemento en la primera posición libre del primer bloque, empezando por el final, salvo en caso en que la primera posición del bloque ya estuviese ocupada, donde sería necesario realizar la petición de un nuevo bloque.

```
ALT1
bq0:  # # # # # 9 8 7 6 5
bq1:  A B C D E F G H I J
bq2:  K L M N O P Q R S T
bq3:  U V W X Y Z 0 1 2 3
bq4:  4 # # # # # # # # #

ALT2
bq0:  # # # # # # # #  9 8
bq1:  7 6 5 A B C D E F G
bq2:  H I J K L M N O P Q
bq3:  R S T  U V W X Y Z 0
bq4:  1 2 3 4 # # # # # #

ALT3
bq 1:  # # # 9 8 7 6 5 A B
bq 2:  C D E F G H I J  K L
bq 3:  M N O P Q R S T U V
bq 4:  W X Y Z 0 1 2 3 4 #
```

1.4.3. Gestión de los bloques

Para poder trabajar con todos los bloques del deque necesitaremos de algun mecanismo para su gestión. En concreto en la práctica utilizaremos una lista (señalar que no es lo más eficiente, sin embargo nos permitirá trabajar sobre nuestro segundo objetivo, utilizar el TDA list), llamemos L a dicha lista. Además de la lista de bloques, necesitaremos de tres atributos más, uno que nos indique cual es el elemento al principio del deque (ini) , el primer elemento válido en el bloque 0, el segundo que nos indicase el final del deque (fin), el siguiente al último, que sería una posición del último bloque. El último nos indicará el número de elementos del deque. Por tanto, la estructura final es

```
class deque {
 public:
   ....
 private:
  #define TAM_BLQ 10
  list<vector<T> > L_dq;
  vector<T>::size_type inicio;  // Inicio de los datos
  vector<T>::size_type final;   // fin de los datos
  size_type tama;
};
```

Así, nuestros deque ejemplos quedarían como

```
ALT1
ini = 5
fin = 1
tama = 36
L= [
  bq0:   # # # # # 9 8 7 6 5
  bq 1:  A B C D E F G H I J
  bq 2:  K L M N O P Q R S T
  bq 3:  U V W X Y Z 0 1 2 3
  bq4:   4 # # # # # # # # #
]

ALT2
ini = 8
fin = 4
tama = 36
L= [
   bq0:   # # # # # # # #  9 8
   bq 1:  7 6 5 A B C D E F G
   bq 2:  H I J K L M N O P Q
   bq 3:  R S T  U V W X Y Z 0
   bq4:   1 2 3 4 # # # # # #
]

ALT3
ini = 3
fin = 9
tama = 36
L= [
   bq 1:  # # # 9 8 7 6 5 A B
   bq 2:  C D E F G H I J  K L
   bq 3:  M N O P Q R S T U V
   bq 4:  W X Y Z 0 1 2 3 4 #
]
```

1.4.4. Acceso Aleatorio a los Elementos

Una vez que tenemos la estructura definida, nos preguntamos cómo podemos implementar el acceso aleatorio, esto es, como identificar la posición del elemento i-ésimo del deque. Lo ilustraremos con un ejemplo. Supongamos que tenemos el deque del ejemplo, deque<char> dq, esto es, dq[0] tiene el 9, dq[1] tiene el 8, ..., dq[dq.size()-1] = 4, y queremos acceder al elemento que ocupa la posición 26 del mismo, por ejemplo haciendo

```
cout << dq[26];
dq[26]=x;
```

En este caso, lo ilustraremos considerando que los elementos se encuentran ubicados según nos indica la alternativa ALT3.

El elemento ini se encuentra en la posición 3 del primer bloque, esto indica que hay 7 elementos (TBQ-ini) en el bloque de inicio (los elementos desde dq[0] a dq[6]). Si consideramos los elementos del segundo bloque habríamos avanzado hasta el decimoséptimo elemento (dq[16]). Por tanto, el elemento que buscamos estará en el tercer bloque, en la posición bq3[9].

```
ALT3
ini = 3
```

```
fin = 9
tama = 36
L= [
   bq 1:  # # # 9 8 7 6 5 A B
   bq 2:  C D E F G H I J  K L
   bq 3:  M N O P Q R S T U V
   bq 4:  W X Y Z 0 1 2 3 4 #
]
```

Tras ver el ejemplo, el algoritmo para encontrar la posición de un elemento implica determinar (si procede) el desplazamiento sobre el primer bloque, para después computar cuantos bloques tendremos que avanzar, y finalmente calcular el desplazamiento sobre el último bloque.

1.4.5. Iteradores sobre el deque

Para poder iterar sobre el deque será necesario tener en cuenta dos factores distintos, el primero nos permite saber en que bloque estamos, mientras que el segundo nos permitirá determinar el desplazamiento dentro de cada bloque.

```
class deque{
....
  class iterator {
    ....
    iterator & operator++();
    ......
    private:
      list<vector<T> >::iterator it_l;
      vector<T>::iterator it_v;
   };
};
```

En este caso, el begin() del deque, it_l apuntará al primer bloque e it_v apuntará a la primera posición que contiene un elemento en dicho bloque. De forma análoga, end() del deque hace que il_l apunte al ultimo bloque e it_v apuntará a la posición siguiente al último elemento en dicho bloque.

Avanzar el iterador implica avanzar a la siguiente posición del deque, que será el siguiente elemento del bloque si no se ha llegado al final del mismo, y en este caso debemos desplazarnos al primer elemento del siguiente bloque.

1.4.6. Inserción de elementos en posiciones intermedias

El proceso es similar, lo que ocurre es que BAJO NINGUNA CIRCUNSTANCIA un bloque debe de tener más de TBQ elementos. Por tanto será necesario desplazar los elementos (hacia adelante o hacia atrás) por el deque. Así, imaginemos que queremos insertar una x justo antes de P . Si consideramos el deque siguiente, deberíamos considerar un iterador a esta posición, esto es, debería apuntar al cuarto elemento del bq 3.

```
ini = 3
```

```
fin = 9
tama = 36
L= [
   bq 1:  # # # 9 8 7 6 5 A B
   bq 2:  C D E F G H I J  K L
   bq 3:  M N O P Q R S T U V
   bq 4:  W X Y Z 0 1 2 3 4 #
]
```

En este caso, debemos de desplazar los elementos del bloque 3 una posición a la derecha, desbordándose el último de ellos, que habrá de posicionarse como primer elemento del siguiente bloque. Este proceso de desbordamiento, se repite en cadena hacia el final del deque. En caso de que el último bloque estuviese lleno, será necesario el solicitar un nuevo bloque al sistema operativo, donde ubicaríamos el último elemento. Finalmente, se ajustaría la posición fin del deque e incrementaríamos en uno el tamaño. Por tanto, nuestro deque quedaría como se ilustra.

```
ini = 3
fin = 10
tama = 37
L= [
   bq 1:  # # # 9 8 7 6 5 A B
   bq 2:  C D E F G H I J  K L
   bq 3:  M N O x P Q R S T U
   bq 4:  V W X Y Z 0 1 2 3 4
]
```

1.5. SE PIDE

En concreto se pide implementar todos los métodos asociados al deque junto con sus iteradores. Además, se debe hacer un análisis empírico de la eficiencia de la implementación. Para ello podeís utilizar el fichero chronos.cpp que está subido a la plataforma decsai.

1.5.1. A ENTREGAR

El alumno debe entregar los siguientes ficheros, con las correciones necesarias para poder trabajar

- deque.h Especificación del TDA diccionario.
- deque.hxx segunda versión del diccionario.
- prueba.cpp fichero de prueba del diccionario
- eficiencia.pdf

Dicha entrega se debe realizar antes de el Lunes 20 de Diciembre, a las 20:00 horas (pm).

Capítulo 2

Índice de clases

2.1. Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

Capítulo 3

Documentación de las clases

3.1. Referencia de la Clase deque::const_iterator

Métodos públicos

- **const_iterator** (const deque< T >::const_iterator &it)
- bool **operator!=** (const deque< T >::const_iterator &it) const
- const T & **operator*** ()
- const_iterator & **operator++** ()
- const_iterator & **operator--** ()
- const_iterator & **operator=** (const deque< T >::const_iterator &it)
- bool **operator==** (const deque< T >::const_iterator &it) const

Atributos privados

- list< vector< T > >::const_iterator **it_l**
- vector< T >::const_iterator **it_v**

Amigas

- class **deque**

La documentación para esta clase fue generada a partir del siguiente fichero:

- deque.h

3.2. Referencia de la Clase deque

```
#include <deque.h>
```

Clases

- class const_iterator
- class iterator

Tipos públicos

- typedef unsigned int **size_type**
- typedef T **value_type**

Métodos públicos

- T & at (size_type pos)

  acceso directo seguro, chequea que la posición sea correcta
- const T & at (size_type pos) const

  acceso directo seguro, chequea que la posición sea correcta
- iterator begin ()

  inicio del deque
- const_iterator begin () const

  inicio del deque
- void clear ()

  elimiar todos los elementos
- deque ()

  Constructor por defecto.
- deque (size_type n, const T &t=T())

  Constructor primitivo.
- deque (const deque< T > &org)

  Constructor de copia.
- bool empty () const

  vacia Chequea si el deque esta vacio (size()==0)
- iterator end ()

  fin del deque,
- const_iterator end () const

  fin del deque,
- iterator erase (typename deque< T >::iterator &pos)

  borrado de elementos
- iterator insert (typename deque< T >::iterator &pos, const T &t)

  Insertar elemento.
- deque< T > & operator= (const deque< T > &org)

  operador de asignacion
- T & operator[] (size_type pos)

  acceso directo
- const T & operator[] (size_type pos) const

acceso directo

- void pop_back (const T &t)

  borrar final
- void pop_front ()

  borrar inicio
- void push_back (const T &t)

  insertar final
- void push_front (const T &t)

  insertar inicio
- void resize (size_type n, const T &t=T())

  Modificar el tamaño.
- size_type size () const

  tamaño Devuelve el numero de elementos en el deque
- void swap (deque< T > &otro)

  intercambia el contenido de dos deques
- ~deque ()

  Destructor. Destruye el receptor liberando los recursos que ocupaba.

Métodos privados

- void **cheq_rep** () const

Atributos privados

- vector< T >::size_type **final**
- vector< T >::size_type **inicio**
- list< vector< T > > **L_dq**
- size_type **tama**

3.2.1. Descripción detallada

deque<T>

deque<T>:: deque, size, capacity, empty, operator[], at, push_back, push_front, resize, operator=, insert, erase,

deque<T>::iterator:: iterator,operator*, operator++, operator-- operator=, operator==, operator!=

deque<T>::const_iterator:: const_iterator,operator*, operator++, operator-- operator=, operator==, operator!=

Descripción

Un deque es un contenedor que permite el acceso aleatorio (mediante valores enteros) a los elementos, inserción y borrado en tiempo constante si se realiza al final y comienzo del deque. Sin embargo, la inserción y el borrado son de orden lineal si se realizan

en posiciones intermedias del deque. Si sólo necesitamos añadir/borrar elementos por el final es preferible utilizar el vector.

Al contrario que los vectores, un deque no garantiza que los elementos se almacenen en posiciones contiguas de memoria, sino que estos se almacenan en bloques de tamaño fijo.

Ambos, vectores y deques, proporcionan una interfaz muy similar y pueden utilizarse para resolver los mismos problemas, pero internamente son muy diferentes. Mientras que los vectores almacenan la informacion de forma contigua, tienendose que redimensionar todo el vector cuando crece su tamaño, los elementos en un deque se pueden almacenan en diferentes en bloques, manteniendo el deque la informacion necesaria para permitir el acceso directo a cualquiera de sus elementos en orden constante. Por tanto tienen una representacion interna mas compleja que el vector, lo que le permite crecer de forma más eficiente bajo ciertas circunstancias, particularmente cuando se consideran secuencias muy grandes de elementos, donde realojarlos todos tiene un elevado coste para el vector. En cualquier caso, la interfaz de un deque permite acceder a los elementos secuencialmente.

La eficiencia de las operaciones de un deque son:

Acceso Aleatorio - O(1). En nuestra implementacíon será un poco más lento Inserción y borrado al principio y al final - O(1), análisis amortizado Inserción y borrado en cualquier otra posición - lineal O(n) El número de elementos en el deque puede variar dinámicamente; la gestión de la memoria es automática.

3.2.2. Documentación del constructor y destructor

3.2.2.1. deque::deque (size_type *n,* const T & *t* = `T()`)

Constructor primitivo.

Parámetros

`in`	*n*	número de elementos en el deque
`in`	*t*	elemento a insertar, por defecto T()

Crea un deque con n elementos con el valor t

3.2.2.2. deque::deque (const deque< T > & *org*)

Constructor de copia.

Parámetros

`in`	*org*	deque que se copia

Crea un deque duplicado exacto de org.

3.2.3. Documentación de las funciones miembro

3.2.3.1. T& deque::at (size_type *pos*)

acceso directo seguro, chequea que la posición sea correcta

Parámetros

in	*pos*	posicion del deque

Devuelve

Devuelve una referencia al elemento n-esimo del deque

3.2.3.2. const T& deque::at (size_type *pos*) const

acceso directo seguro, chequea que la posición sea correcta

Parámetros

in	*pos*	posicion del deque

Devuelve

Devuelve una referencia cosntnte al elemento n-esimo del deque

3.2.3.3. void deque::clear ( )

elimiar todos los elementos

Elimina todos los elementos del contenedor.

3.2.3.4. iterator deque::end ( )

fin del deque,

devuelve la posición siguiente al último elemento

3.2.3.5. const_iterator deque::end ( ) const

fin del deque,

devuelve la posición siguiente al último elemento

3.2.3.6. iterator deque::erase (typename deque< T >::iterator & *pos*)

borrado de elementos

Parámetros

in	*pos*	posicion a borrar

Borra el elemento de la posición pos, desplaza los siguientes elementos una posición hacia el inicio del deque

3.2.3.7. iterator deque::insert (typename deque< T >::iterator & *pos,* const T & *t*)

Insertar elemento.

Parámetros

in	*pos*	posicion a insertar
in	*T*	elemento a insertar

Inserta el elemento t justo antes de pos, desplaza el resto de los elementos una posición hacia el final del deque

3.2.3.8. deque<T>& deque::operator= (const deque< T > & *org*)

operador de asignacion

Parámetros

org]	el deque a asignar.

3.2.3.9. T& deque::operator[] (size_type *pos*)

acceso directo

Parámetros

in	*pos*	posicion del deque

Devuelve

Devuelve una referencia al elemento n-esimo del deque

3.2.3.10. const T& deque::operator[] (size_type *pos*) const

acceso directo

Parámetros

in	*pos*	posicion del deque

Devuelve

Devuelve una referencia cosntnte al elemento n-esimo del deque

3.2.3.11. void deque::pop_back (const T & *t*)

borrar final

Elimina el elemento al final del deque

3.2.3.12. void deque::pop_front ( )

borrar inicio

Elimina el elemento al inicio del deque

3.2.3.13. void deque::push_back (const T & *t*)

insertar final

Parámetros

in	*t*	elemento a insertar del deque

Añade el elemento t al final del deque

3.2.3.14. void deque::push_front (const T & *t*)

insertar inicio

Parámetros

in	*t*	elemento a insertar del deque

Añade el elemento t al final del deque

3.2.3.15. void deque::resize (size_type *n,* const T & *t* = `T()`)

Modificar el tamaño.

Parámetros

in	*n*	número de elementos en el deque
in	*t*	elemento a insertar, por defecto T()

Si el numero de elementos es menor que n, se añaden al final copias de t, hasta alcanzar un tamaño n. Si el numero de elementos es mayor que n, el deque se reduce a los primeros n elementos.

3.2.3.16. void deque::swap (deque< T > & *otro*)

intercambia el contenido de dos deques

Intercambia el contenido entre *this y otro, No implica borrado o copia de elementos

La documentación para esta clase fue generada a partir del siguiente fichero:

- deque.h

3.3. Referencia de la Clase deque::iterator

Métodos públicos

- **iterator** (const deque< T >::iterator &it)
- bool **operator!=** (const deque< T >::iterator &it) const
- T & **operator*** ()
- iterator & **operator++** ()
- iterator & **operator--** ()
- iterator & **operator=** (const deque< T >::iterator &it)
- bool **operator==** (const deque< T >::iterator &it) const

Atributos privados

- list< vector< T > >::iterator **it_l**
- vector< T >::iterator **it_v**

Amigas

- class **deque**

La documentación para esta clase fue generada a partir del siguiente fichero:

- deque.h