Standard Template Library

Bach. Rodolfo Mercado Gonzales
Programación Competitiva UPC

Introducción

“ Es mejor reutilizar que reescribir ”

Standard Template Library (STL)

- Librería de algoritmos y estructuras de datos que forman parte del estándar de C++.
- Evita que se tengan que programar constantemente algoritmos y estructuras de uso frecuente.
- La STL internamente tiene una implementación realmente compleja, sin embargo nos permite usarla de una manera extremadamente sencilla.

Componentes

- **Contenedores**

  Estructuras de datos que almacenan elementos del mismo tipo.

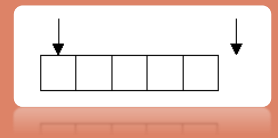
- **Algoritmos**

  Funciones parametrizadas.

- **Iteradores**

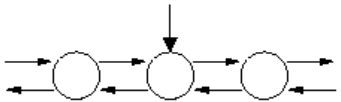
  Punteros usados en los contenedores y algoritmos.

Iteradores



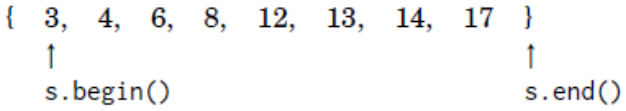
Iteradores

- Es una variable que apunta a un elemento en un contenedor (puntero).
- Nos permite acceder y recorrer los elementos de un contenedor.
- Son usados como parámetros en muchas de las funciones de la STL.



Iteradores

- Iterador que apunta al primer elemento del contenedor — contenedor.begin()
- Iterador a una posición después del último elemento — contenedor.end()

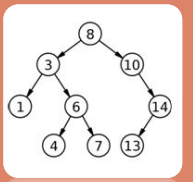


Iteradores

- Apuntar al siguiente elemento iterador++
- Apuntar el elemento anterior iterador--
- Acceder al valor del elemento apuntado *iterador
- Un rango es un par de iteradores apuntando al inicio y a una posición después del final de una subsecuencia de un contenedor. [iterador_inicio, iterador_fin >

```
reverse(v.begin(), v.end());
```

Contenedores



Contenedores

- Estructuras que pueden almacenar una colección de elementos del mismo tipo.
- Se accede a sus elementos a través de iteradores.
- Administra su memoria en el heap.

en C solo existía un tipo de contenedor: arreglo

Contenedores

- **Contenedores de secuencia**

  El usuario controla el orden de los elementos.

  vector y deque.

- **Contenedores asociativos**

  El contenedor controla la posición de los elementos, manteniendo un determinado orden.

  set, multiset, map y multimap

- **Adaptadores de contenedores**

  Contenedores implementados en base a otros.

  stack, queue y priority_queue

Vector

- Contenedor de secuencia que representa un arreglo que puede cambiar de tamaño.
- Permite agregar y eliminar eficientemente un elemento en el final del contenedor (tiempo constante).
- Almacena elementos en posiciones contiguas de memoria.
- Permite acceso aleatorio.

Vector

```
vector<int> v; //declaración de un vector
v.push_back(x); //agrega un elemento al final en O(1)
v.pop_back(); //elimina el último elemento en O(1)
v.back(); // devuelve el último elemento en O(1)
v.size(); // devuelve el tamaño del vector en O(1)

/*acceder a la posición p*/
v[p]; // O(1)

/*insertar x en la posición p*/
v.insert(v.begin() + p, x); // O(n)

/*borrar elemento de la posición p*/
v.erase(v.begin() + p); // O(n)
```

Ejercicios

- HackerRank – Vector-Sort
- HackerRank – Vector-Erase

Deque

- Contenedor de secuencia parecido al vector.
- Permite agregar y eliminar eficientemente un elemento en el final del contenedor (tiempo constante).
- Permite agregar y eliminar eficientemente un elemento en el inicio del contenedor (tiempo constante).
- Permite acceso aleatorio.
- En general sus operaciones son una constante más lentas que las del vector.

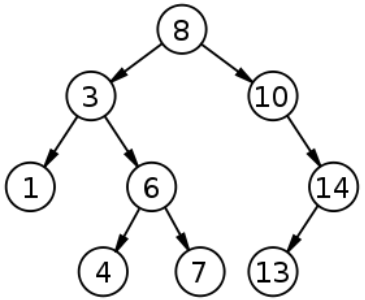
Deque

```
deque<int> dq; //declaración de un vector
dq.push_back(x); //agrega un elemento al final en O(1)
dq.pop_back(); //elimina el último elemento en O(1)
dq.push_front(x); //agrega un elemento al inicio en O(1)
dq.pop_front(); //elimina el primer elemento en O(1)
dq.back(); // devuelve el último elemento en O(1)
dq.size(); // devuelve el tamaño del vector en O(1)

/*acceder a la posición p*/
dq[p]; // O(1)

/*insertar x en la posición p*/
dq.insert(dq.begin() + p, x); // O(n)

/*borrar elemento de la posición p*/
dq.erase(dq.begin() + p); // O(n)
```

Árbol de búsqueda binaria (BST)

Estructura de datos que mantiene un conjunto de elementos en un determinado orden; tal que:

- El subárbol izquierdo de cualquier nodo, contiene valores menores que el valor de dicho nodo.
- El subárbol derecho de cualquier nodo, contiene valores mayores que el valor de dicho nodo.



los contenedores asociativos se basan en un BST de altura logarítmica.

Set

- Contenedor asociativo basado en un árbol de búsqueda binaria balanceado.
- Permite agregar, buscar y eliminar un elemente do manera eficiente (tiempo logarítmico)
- Los elementos son vistos como una llave (key), por ello deben ser únicos.
- Los elementos son almacenados en orden ascendente.

Set

```cpp
set<int> s; //declaración de un set
s.insert(x); //insertar x en O(log n)
auto it = s.find(x); //buscar x en O(log n)
s.count(x); //obtener el número de ocurrencias de x en O(log n)
s.erase(x); //eliminar x en O(log n)
s.erase(it); //eliminar elemento mediante un iterador en O(1)
s.lower_bound(x); //primer elemento >= x en O(log n)
s.upper_bound(x); //primer elemento > x en O(log n)

/*recorrido de un set*/
for (auto e : s) {
    cout << e << "\n";
}
```

Map

- Contenedor asociativo basado en un árbol de búsqueda binaria balanceado.
- Es similar a un set, que nos permite almacenar pares de la forma $< llave, valor >$, donde la llave es única.
- Permite agregar, buscar y eliminar un elemento de manera eficiente (tiempo logarítmico)
- Los elementos son almacenados en orden ascendente (con respecto a la llave).

Map

```
map<string, int> m; //declaración de un map
m.insert({s, x}); //insertar el par {s, x} en O(log n)
m[s] = 18; //insertar o actualiza según sea el caso, en O(log n)
auto it = m.find(s); //buscar la llave s en O(log n)
m.count(s); //obtener el número de ocurrencias de s en O(log n)
m.erase(s); //eliminar s en O(log n)
m.erase(it); //eliminar elemento mediante un iterador en O(1)
m.lower_bound(s); //primer elemento >= x en O(log n)
m.upper_bound(s); //primer elemento > x en O(log n)

/*recorrido de un map*/
for (auto e : m) {
    cout << e.first << " " << e.second << "\n";
}
```

Multiset

- Contenedor asociativo similar al set, que adicionalmente permite guardar elementos repetidos.

```
multiset<int> s; //declaración de un multiset
s.insert(x); //insertar x en O(log n)
auto it = s.find(x); //buscar x en O(log n)
s.count(x); //obtener el número de ocurrencias de x en O(log n + k)
s.erase(x); //eliminar todos los x en O(log n + k)
s.erase(it); //eliminar elemento mediante un iterador en O(1)
s.lower_bound(x); //primer elemento >= x en O(log n)
s.upper_bound(x); //primer elemento > x en O(log n)

/*recorrido de un multiset*/
for (auto e : s) {
    cout << e << "\n";
}
```

Multimap

- Contenedor asociativo similar al map, que adicionalmente permite guardar llaves repetidas.

```
multimap<string, int> m; //declaración de un multimap
m.insert({s, x}); //insertar el par {s, x} en O(log n)
auto it = m.find(s); //buscar la llave s en O(log n)
m.count(s); //obtener el número de ocurrencias de s en O(log n + k)
m.erase(s); //eliminar todos los s en O(log n + k)
m.erase(it); //eliminar elemento mediante un iterador en O(1)
(*it).second = x; //actualizar valor mediante un iterador en O(1)
m.lower_bound(s); //primer elemento >= x en O(log n)
m.upper_bound(s); //primer elemento > x en O(log n)

/*recorrido de un multimap*/
for (auto e : m) {
    cout << e.first << " " << e.second << "\n";
}
```

Ejercicios

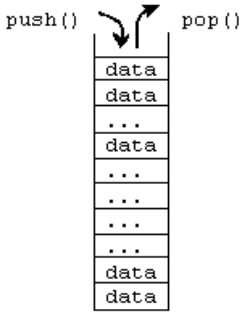
- HackerRank – Sets-STL
- HackerRank – Maps-STL
- Codeforces – Is your horseshoe on the other hoof?
- UVA 10420 – List of Conquests

Desafíos

- Codechef – Forgotten Language
- Codeforces – Restoring Password
- Codeforces – Radio Station
- Codeforces – Registration System
- Codechef – Missing a Point
- AtCoder – Good Sequence
- Hackerrank – Minimum Loss
- Live Archive – Sum the Square
- Codechef – Stacks

Stack (pila)

- Estructura de datos en la que las operaciones realizada siguen el orden LIFO (last in - first out).
- La implementación esta basada en un deque.



Stack (pila)

```
stack<int> p; //declaración de una pila
p.push(x); //inserta elemento al final en O(1)
p.pop(); //elimina el último elemento en O(1)
p.top(); //retorna el último elemento en O(1)
p.empty(); //verifica si la pila está vacía en O(1)
p.size(); //retorna el tamaño de la pila en O(1)
```

Paréntesis Balanceados

Dado una cadena compuesta de paréntesis, es decir ‘(‘ y ‘)’ , indicar si los paréntesis se encuentran balanceados (cada paréntesis de apertura tiene su cierre correspondiente y todos están correctamente anidados)

Ejemplos:

( ) si

(( ) ( )) si

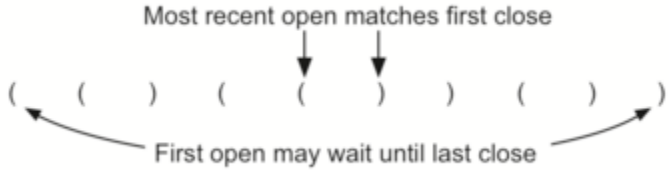
(( )) (no

Paréntesis Balanceados

Hint:

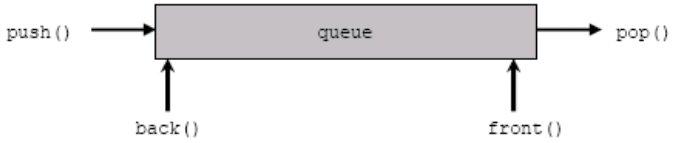
Si procesamos los paréntesis de izquierda a derecha:

- El más reciente de apertura debe hacer match con el más próximo de cierre.
- El primer paréntesis de apertura puede esperar hasta el último de cierre.



Queue (cola)

- Estructura de datos en la que las operaciones realizada siguen el orden FIFO (first in - first out).
- La implementación esta basada en un deque.



Queue (cola)

```
queue<int> q; //declaración de una cola
q.push(x); //inserta elemento al final en O(1)
q.pop(); //elimina el primer elemento en O(1)
q.front(); //retorna el primer elemento en O(1)
q.empty(); //verifica si la cola está vacía en O(1)
q.size(); //retorna el tamaño de la cola en O(1)
```

Priority queue (cola de prioridad)

- Contenedor basado en una estructura conocida como heap.
- Permite insertar un elemento de manera eficiente.
- Permite recuperar y eliminar el máximo elemento de manera eficiente.

Priority queue (cola de prioridad)

```
priority_queue<int> pq; //declaración de una cola de prioridad
pq.push(x); //inserta un elemento en O(log n)
pq.top(); //retorna el mayor elemento O(1)
pq.pop(); //elimina el mayor elemento O(log n)
pq.empty(); //verifica si el contenedor esta vacío en O(1)
pq.size(); //retorna le tamaño del contenedor en O(1)
```

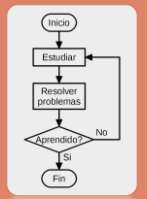
Ejercicio

- Codeforces – Plug-in

Desafíos

- Hackerrank - Balanced Brackets
- Hackerrank - Maximum element
- Hackerrank - Equal Stacks
- UVA - Throwing cards

Algoritmos



Sort

Permite ordenar eficientemente un rango $[inicio, fin >$ de un contenedor de secuencia (también funciona en arreglos).

```
sort(v.begin(), v.end()); //vector o deque
sort(A, A + n); //arreglo
```

$complejidad: O(n \log n)$

Binary search

- Permite saber de manera eficiente si un elemento x está presente en un rango $[\boldsymbol{inicio}, \boldsymbol{fin} >$ de un contenedor de secuencia (también funciona en arreglos).
- Como requisito el contenedor debe estar ordenado.

```
bool ok1 = binary_search(v.begin(), v.end(), x); //vector o deque
bool ok2 = binary_search(A, A + n, x); //arreglo
```

$complejidad$: $O(\log n)$

Lower bound

- Retorna un iterador que apunta al primer elemento de un rango $[\boldsymbol{inicio}, \boldsymbol{fin} >$, de un contenedor de secuencia, que es mayor o igual a x (también funciona en arreglos).
- Como requisito el contenedor debe estar ordenado.

```
auto it = lower_bound(v.begin(), v.end(), x); //vector o deque
auto p = lower_bound(A, A + n, x); //arreglo
```

$complejidad$: $O(\log n)$

Upper bound

- Retorna un iterador que apunta al primer elemento de un rango $[inicio, fin >$, de un contenedor de secuencia, que es mayor a x (también funciona en arreglos).
- Como requisito el contenedor debe estar ordenado.

```
auto it = upper_bound(v.begin(), v.end(), x); //vector o deque
auto p = upper_bound(A, A + n, x); //arreglo
```

$complejidad$: $O(\log n)$

Desafíos

- SPOJ – Búsqueda binaria
- SPOJ – Búsqueda binaria 2

Referencias

- University of Helsinski - Algorithm Libraries
- Antti Laaksonen - Guide to Competitive Programming
- Topcoder – Power up C++ with the Standard Template Library
- Omer Giménez - Guía de Programación C++ STL
- cpluplus.com - Containers

“The road to success is always under construction.”

- Lily Tomlin