Apuntes OIFem 2020

Estructuras de Datos 1

Stacks (pilas)

Una pila, como su propio nombre indica, funciona como un montón de cartas, donde coger "la carta de arriba" equivale a coger la última carta añadida al montón. Este tipo de estructuras de datos se llaman **last-in, first-out (LIFO)**.

A una pila podemos añadirle y quitarle elementos y ver cuál es el elemento que está arriba del todo. Además, tenemos la posibilidad en todo momento de saber cuántos elementos tiene.

Hay que tener en cuenta que todos los elementos de una pila deben de ser del mismo tipo.

Funciones

- Añadir un elemento x a la pila: `pila.push(x);`
- Quitar el último elemento de la pila: `pila.pop();`
- Ver cuántos elementos hay en la pila: `pila.size()`
- Comprobar si la pila está vacía: `pila.empty()`
- Ver cuál es el elemento de encima: `pila.top()`

Todas estas funciones tienen una complejidad de tiempo constante (proporcional a $O(1)$).

Ejemplo

Darle la vuelta a un string

```cpp
#include <iostream>
#include <stack>
using namespace std;

void reverseStr(string & S) {
    stack <char> pila;
    for (int i = 0; i < (int) S.length(); i++) {
        pila.push(S[i]);
    }
    S = "";
    while(! pila.empty()) {
        S.push_back(pila.top());
        pila.pop();
    }
}

int main()
{
    string S;
    cin >> S;
    reverseStr(S);
    cout << S << '\n';
}
```

Queues (colas)

Una cola se asemeja a una fila en un supermercado, donde la primera persona en salir es la primera en haber entrado. Este tipo de estructuras se llaman **first-in, first-out (FIFO)**.

Como a una pila, podemos añadir y quitar elementos y usarla como usaríamos una pila salvo por la diferencia en orden de salida. Además, todos sus elementos deben ser del mismo tipo.

Funciones

- Añadir un elemento x a la cola: `cola.push(x);`
- Quitar el primer elemento de la cola: `cola.pop();`
- Ver cuántos elementos hay en la cola: `cola.size()`
- Comprobar si la cola está vacía: `cola.empty()`
- Ver cuál es el primer elemento: `cola.front()`

Todas estas funciones tienen una complejidad de tiempo constante (proporcional a $O(1)$).

Ejemplo

Números de Hamming (números cuyos únicos factores primos sean 2, 3 y 5)

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

vector<int> hamming(int N) {
    vector<bool> vistos;
    vistos.assign(N+1, false);
    queue<int> nums;
    nums.push(1);
    int x;

    vector<int> numerosHamming = {};
    while(nums.size()) {
        x = nums.front();
        nums.pop();
        if (x > N || vistos[x])
            continue;

        vistos[x] = true;
        if (x > 1)
            numerosHamming.push_back(x);

        nums.push(2*x);
        nums.push(3*x);
        nums.push(5*x);
    }
    sort(numerosHamming.begin(), numerosHamming.end());
    return numerosHamming;
}

int main()
{
    int N;
    cin >> N;
    vector<int> vec = hamming(N);
    for (int num: vec)
        cout << num << '\n';
}
```

Priority queues (montículos)

A simple vista, un montículo funciona como una cola, con la diferencia de que su primer elemento es el mayor elemento de la cola.

Veremos que las diferencias van más allá por cómo están implementadas ambas por debajo. Esto afecta a la complejidad de tiempo de los montículos.

Funciones

n es igual al número de elementos en la estructura.

- Añadir un elemento x al montículo ($O(\log n)$): `monticulo.push(x);`
- Quitar el primer elemento del montículo ($O(\log n)$): `monticulo.pop();`
- Ver cuántos elementos hay en el montículo ($O(1)$): `monticulo.size()`
- Comprobar si el montículo está vacío ($O(1)$): `monticulo.empty()`
- Ver cuál es el mayor elemento ($O(1)$): `monticulo.top()`

Ejemplo

Ordenar una lista en orden descendente

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void ordenar (vector<int> & vec) {
    priority_queue<int> monticulo;
    while(vec.size()) {
        monticulo.push(*vec.rbegin());
        vec.pop_back();
    }
    while(monticulo.size()) {
        vec.push_back(monticulo.top());
        monticulo.pop();
    }
}

int main() {
    int N;
    cin >> N;
    vector<int> vec(N);
    for (int i = 0; i < N; i++)
        cin >> vec[i];
    ordenar(vec);
    for (int i = 0; i < N; i++)
        cout << vec[i] << ' ';
}
```

Extra: utiliza el truco del -1 para alterar la función y que ordene de forma ascendente.

Conjuntos desordenados

Un conjunto desordenado en C++ es un grupo de valores únicos del mismo tipo que sirve para comprobar de forma rápida si ya nos hemos topado con un valor. Todas sus operaciones funcionan en tiempo constante ($O(1)$), las principales siendo añadir y quitar elementos.

Funciones

- Añadir un elemento x al conjunto ($O(1)$): `conjunto.insert(x);`
- Quitar un elemento x del conjunto ($O(1)$): `conjunto.erase(x);`
- Comprobar si el número x está en el conjunto ($O(1)$): `conjunto.find(x) != conjunto.end()`
- Ver cuántos elementos hay en el conjunto ($O(1)$): `conjunto.size()`
- Comprobar si el conjunto está vacío ($O(1)$): `conjunto.empty()`

Ejemplo

Quitar las repeticiones de una lista

```cpp
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;

void numerosUnicos (vector<int> & vec) {
    unordered_set<int> vistos;
    while (vec.size()) {
        vistos.insert(*vec.rbegin());
        vec.pop_back();
    }
    for (auto it = vistos.cbegin(); it != vistos.cend(); it++)
        vec.push_back(*it);
}

int main()
{
    int N;
    cin >> N;
    vector<int> vec(N);
    for (int i = 0; i < N; i++)
        cin >> vec[i];
    numerosUnicos(vec);
    for (int i = 0; i < (int) vec.size(); i++)
        cout << vec[i] << '\n';
}
```

Conjuntos ordenados

Un conjunto ordenado funciona aparentemente como un conjunto, con la ventaja extra de que los elementos están ordenados según un comparador (por defecto es el mismo que utiliza `sort()`). Sin embargo, este orden lo pagas con eficiencia (por debajo está implementado con una estructura arbórea que veremos en breve).

Funciones

n es igual al número de elementos en la estructura.

- Añadir un elemento x al conjunto ($O(\log n)$): `conjunto.insert(x);`
- Quitar un elemento x del conjunto ($O(\log n)$): `conjunto.erase(x);`
- Comprobar si el número x está en el conjunto ($O(\log n)$): `conjunto.find(x) != conjunto.end()`
- Ver cuántos elementos hay en el conjunto ($O(1)$): `conjunto.size()`
- Comprobar si el conjunto está vacío ($O(1)$): `conjunto.empty()`

Ejemplo

Quitar las repeticiones de una lista, dejándola ordenada

```cpp
#include <iostream>
#include <vector>
#include <set>
using namespace std;

void numerosUnicos (vector<int> & vec) {
    set<int> vistos;
    while (vec.size()) {
        vistos.insert(*vec.rbegin());
        vec.pop_back();
    }
    for (auto it = vistos.cbegin(); it != vistos.cend(); it++)
        vec.push_back(*it);
}

int main()
{
    int N;
    cin >> N;
    vector<int> vec(N);
    for (int i = 0; i < N; i++)
        cin >> vec[i];
    numerosUnicos(vec);
    for (int i = 0; i < (int) vec.size(); i++)
        cout << vec[i] << '\n';
}
```

Mapas desordenados

Un mapa desordenado es un conjunto de parejas etiqueta-valor, donde una serie de elementos del tipo A tienen su correspondencia en el tipo B. Los mapas desordenados son eficientes, con sus funciones operando en tiempo constante ($O(1)$).

Nota: tanto los conjuntos como los mapas desordenados empiezan con un tamaño de 1, amplían a uno de 2 cuando lo ven necesario y sucesivamente, duplicando su tamaño. Si ya sabes de antemano que vas a necesitar bastante más espacio que eso, es útil avisar al compilador, ya que cada ampliación de esas es cara. Sobre todo al principio, ya que las potencias de 2 están muy pegadas unas a otras. Esto se hace a través de la función `reserve()`, donde puedes reservar ya cierto espacio con antelación. Esto será más rápido si reservas una potencia de 2. Yo suelo reservar 7192, por ejemplo. Consiste en, debajo de la línea donde declaras el conjunto/mapa, añadir un `mapa.reserve(7192);`.

Funciones

n es igual al número de elementos en la estructura.

- Añadir una pareja $a \to b$ ($O(1)$): `mapa[a] = b;`
- Quitar una etiqueta a del mapa ($O(1)$): `mapa.erase(a);`
- Comprobar si la etiqueta a está en el mapa ($O(1)$): `mapa.find(a) != mapa.end()`
- Ver cuántos elementos hay en el mapa ($O(1)$): `mapa.size()`
- Comprobar si el mapa está vacío ($O(1)$): `mapa.empty()`

Ejemplo

Lista de contactos de un teléfono móvil

```
#include <unordered_map>
using namespace std;
typedef long long int ll;

void anadirContacto(string & nombre, ll numero, unordered_map<string, ll> &
↪  libretaDeContactos) {
    libretaDeContactos[nombre] = numero;
}

ll numeroDelContacto(string & nombre, unordered_map<string, ll> &
↪  libretaDeContactos) {
    return libretaDeContactos[nombre];
}

void borrarContacto(string & nombre, unordered_map<string, ll> &
↪  libretaDeContactos) {
    libretaDeContactos.erase(nombre);
}
```

Mapas ordenados

Un mapa ordenado funciona aparentemente como un mapa, con la ventaja extra de que los elementos están ordenados según un comparador (por defecto es el mismo que utiliza `sort()`). Sin embargo, este orden lo pagas con eficiencia (por debajo está implementado con una estructura arbórea que veremos en breve).

Funciones

n es igual al número de elementos en la estructura.

- Añadir una pareja $a \to b$ ($O(\log n)$): `mapa[a] = b;`
- Quitar una etiqueta a del mapa ($O(\log n)$): `mapa.erase(a);`
- Comprobar si la etiqueta a está en el mapa ($O(\log n)$): `mapa.find(a) != mapa.end()`
- Ver cuántos elementos hay en el mapa ($O(1)$): `mapa.size()`
- Comprobar si el mapa está vacío ($O(1)$): `mapa.empty()`

Ejemplo

Nombres y edades en orden alfabético

```cpp
#include <iostream>
#include <map>
using namespace std;

int main()
{
    int N, edad;
    cin >> N;
    map<string, int> concursantes;
    string nombre;
    for (int i = 0; i < N; i++) {
        cin >> nombre;
        cin >> edad;
        concursantes[nombre] = edad;
    }
    for (auto it = concursantes.cbegin(); it != concursantes.cend(); it++)
        cout << it->first << " concursa en la OIFem y tiene " << it->second << "
        ↪  años\n";
}
```