Preparazione alle Olimpiadi di Informatica

UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Strutture dati della STL: array dinamici, insiemi, mappe e iteratori

Mario Alviano

Strutture dati

- Come conviene memorizzare le informazioni?
- Quali operazioni sono efficienti con una struttura dati?
- La libreria standard ha già una struttura dati per il mio problema?
- Vediamo le più comuni

Array dinamico

- Un array di dimensione variabile
- Esempi: std::vector<T>, std::string
- Accesso a elemento i-esimo in O(1)
- Inserimento ed eliminazione in O(n)
- Inserimento in coda in O(1) (a meno di riallocazione)
- Eliminazione in coda in O(1)

std::vector (1)

- Dichiarazione: vector<int>, vector<double>, …
- Metodi comuni:
  - size(), empty()
  - operator[](indice)
  - push_back(elemento)
  - pop_back()
  - back()

std::vector (2)

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

```
vector<int> v = {2,4,2,5,1};
```

```
for (auto x : v) {
    cout << x << "\n";
}
```

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

std::string (1)

- Dichiarazione: string
- Metodi comuni:
  - size(), empty()
  - operator[](indice)
  - operator+(string)
  - substr(indice, lunghezza)
  - find(sottostringa)

std::string (2)

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

std::stack (1)

- Last in, first out (LIFO)
- Dichiarazione: stack<int>, stack<double>, …
- Metodi comuni:
  - push(elemento)
  - pop()
  - top()
- Essenzialmente i metodi *back di vector
- Lavora in O(1) (a meno di riallocazione)

std::stack (2)

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

std::queue (1)

- First in, first out (FIFO)
- Dichiarazione: queue<int>, queue<double>, …
- Metodi comuni:
  - push(elemento)
  - pop()
  - front()
- Essenzialmente i metodi push_front(), pop_back() e back() di deque
- Lavora in O(1) (a meno di riallocazioni)

std::queue (2)

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

std::deque (1)

- Inserimenti in testa e in coda in O(1)
- Più lento di un vector nelle altre operazioni
- Dichiarazione: deque<int>, deque<double>, …
- Metodi comuni:
  - push_back(elemento), pop_back(), back()
  - push_front(elemento), pop_front(), front()

std::deque (2)

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

std::priority_queue (1)

- Gli elementi hanno una priorità (saltano la coda)
- Implementato come **heap** nell'STL
  - Inserimento e rimozione in O(log n)
- Dichiarazione: priority_queue<int>, priority_queue<int, vector<int>, greater<int>>
- Metodi comuni:
  - push(elemento), top(), pop()

std::priority_queue (2)

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

```
priority_queue<int,vector<int>,greater<int>> q;
```

std::bitset (1)

- Array dinamico di bit
- Molto efficiente in memoria (usa i bit)
- Dichiarazione: bitset<dimensione>
- Metodi comuni:
  - costruttore da string
  - operator[]
  - count()
  - bit operators: &, |, ^

std::bitset (2)

```
bitset<10> s(string("0010011010")); // from right to left
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

Insiemi (1)

- Strutture dati con elementi **distinti** (nessuna ripetizione)
- Due implementazioni in STL
  - set: albero binario bilanciato, mantiene l'ordine, lavora in O(log n)
  - unordered_set: hashmap, non mantiene l'ordine, lavora in O(1)
- Metodi comuni:
  - size(), empty()
  - insert(elemento), erase(elemento)
  - count(elemento), find(elemento)

Insiemi (2)

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

Multiinsiemi (1)

- Essenzialmente set + un contatore per ogni elemento
- Due implementazioni in STL
  - multiset
  - unordered_multiset
- Metodi analoghi a set e unordered_set
  - count() non ritorna più solo 0 o 1
  - erase(elemento) vs erase(iteratore)

Multiinsiemi (2)

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Mappe (1)

- Essenzialmente set di **chiavi** + un **valore** per ogni chiave
- Due implementazioni in STL
  - map: chiavi ordinate, lavora in O(log n)
  - unordered_map: chiavi non ordinate, lavora in O(1)
- Dichiarazione: map<string, int>, map<int, int>, …
- Metodi comuni:
  - size(), empty()
  - operator[]
  - insert(make_pair(chiave, valore)), erase(chiave)
  - find(chiave)

Mappe (2)

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

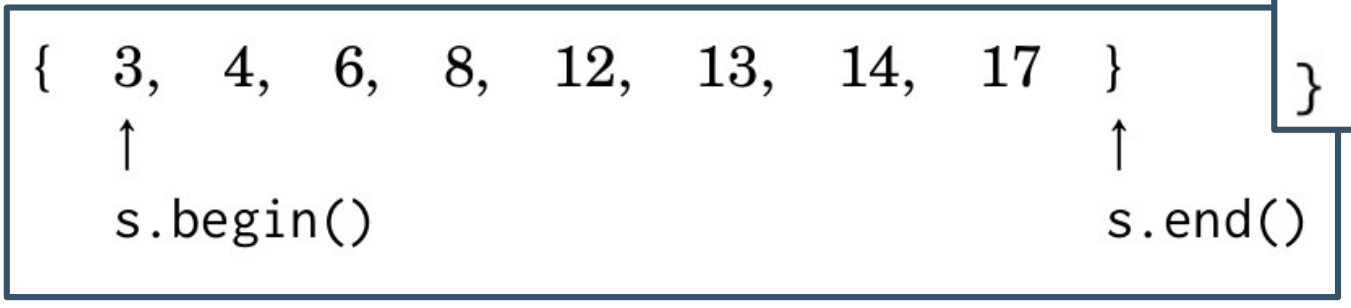
```
if (m.count("aybabtu")) {
    // key exists
}
```

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

Iteratori (1)

- Variabili che **puntano** elementi in una struttura dati
- Molte strutture dati forniscono i metodi begin() ed end()
- Metodi di ricerca restituiscono end() se non trovano l'elemento richiesto

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```



Iteratori (2)

- Molte funzioni STL usano iteratori
  - Per indicare un intervallo di elementi
  - Per iterare
  - Come output di un metodo di ricerca

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

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Iteratori (3)

- Dichiarazione: set<int>::iterator, vector<int>::iterator, …
- Metodi comuni:
  - operator*
  - operator++, operator--
- Esistono anche i reverse_iterator:
  - set<int>::reverse_iterator()
  - rbegin() e rend() per ottenere un reverse_iterator

Iteratori su insiemi (1)

Minimo

```
auto it = s.begin();
cout << *it << "\n";
```

Massimo

```
auto it = s.end(); it--;
cout << *it << "\n";
```

Si può usare anche rbegin():

cout << *s.rbegin() << endl;

Iterazione ordinata

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

Iteratori su insiemi (2)

- set ha metodi lower_bound() e upper_bound()
  - Simili alle funzioni omonime

```
set<int> A = {1, 2, 6, 9, 7, 3, 5, 8};
print(A);

{
    auto res = A.lower_bound(4);
    cout << "first element >= 4 is " << *res << endl;
}
{
    auto res = A.upper_bound(5);
    cout << "first element >  5 is " << *res << endl;
}
```

Esercizi suggeriti

- Department blackout (recovery)
  - Ordinare i voti
  - Usare unordered_multiset per verificare velocemente se una sequenza esiste
- Saddest friend (maxim)
  - Usare multiset per avere massimo e minimo in modo efficiente
- Parser HTML (html)
  - Singola iterazione sulla stringa di input
- Gambling Assistant (gamble)
  - Usare priority_queue per scartare la carta di valore minimo
- Desk ordering (desk)
  - Usare priority_queue per trovare primo e secondo massimo
- Playing with barrels (barrels)
  - Array che indica il barile in cui defluisce ogni barile (skip list)
- Game Simulator (dominion)
  - Usare deque per il deck e una mappa per i prezzi (ordinati dal più caro al più economico)

Fine della lezione