情報理工学基礎 1

期末レポート

学籍番号: 25B30529

名前: 小林颯

1. 概要

本レポートでは、授業で扱われた概念が C++ と Python でどのように実装できるかを述べる。これらの言語を私が選択したのは、大学に入ってから始めた競技プログラミングを通じて C++ に習熟したこと、また高校時代から親しんできた Python をより活用したいと考えたからである。

2. 実装

この章では、授業で扱われた概念が C++ と Python でどのように実装できるかを具体的に述べる。特に、C++ ではコード例を示すにあたり、すべての実装において以下の共通設定を前提とする。

```
#include <bits/stdc++.h>
using namespace std;
```

また、ユーザー定義関数を用いない場合はmain関数の中身のみを示すことにする。これは`std::` を省略することにより、コードをより簡潔に記述することを目的としている。

2.1. 集合

この章では集合に関する概念を実装する。

2.1.1. 集合(有限集合)

有限集合は要素数が有限である集合である。C++では`set`を用いて直接実装することができる。

```
set<int> mySet = {1, 1, 2, 2, 3, 3, 3, 4, 5};
println("mySet is: {}", mySet);
//output mySet is: {1, 2, 3, 4, 5}
```

2.1.2. 要素

要素は数である必要はなく、集合や文字列であってもよい。

```
set<set<int>> set_1 = {{1, 2}, {3, 4}, {3, 4} ,{5, 6}};
println("set is: {}", set_1);
//output set_1 is: {{1, 2}, {3, 4}, {5, 6}}
```

```
set<string> set_2 = {"hello", "world", "hello", "world", "!"};
println("set_2 is: {}", set_2);
//output set_2 is: {"!", "hello", "world"}
```

2.1.3. 空集合

要素が一つもない集まりも集合であり、これを空集合と呼ぶ。具体的な要素を記述しないことにより、空集合を表現することができる。集合が空集合であるかどうかは C++ では `std::set` のメンバ関数 `empty()` を用いることで確認することができる。

```
set<int> mySet = {};
if(mySet.empty()) {
    printf("%s\n", "mySet is empty");
} else {
    printf("%s\n", "mySet is not empty");
}
//output mySet is empty

mySet.insert(1);
if(mySet.empty()) {
    printf("%s\n", "mySet is empty");
} else {
    printf("%s\n", "mySet is not empty");
}
//output mySet is not empty
```

2.1.4. 要素数

C++ では要素数は `std::set` のメンバ関数 `size()` を用いることで取得することができる。

```
set<int> mySet = {1, 2, 3, 4, 5};
printf("%s %d\n","mySet size is:", mySet.size());
//output mySet size is: 5
```

2.1.5. 無限集合

コンピュータのメモリは有限であるため、無限に多くの要素を持つ集合を直接的に保持することはできない。しかし、C++ではコルーチン機能を使用することにより、ジェネレータという形で無限集合の概念を表現することができる。ジェネレータは必要に応じて次の要素を生成する遅延評価の仕組みであり、理論上無限の要素を扱うことが可能である。以下では、授業内で扱った無限集合をジェネレータを用いて実装する例を示す。

2.1.6. 自然数

数学における、$\mathbb{N} = 1,2,3, \ldots$ は、無限に多くの要素を持つ集合である。コンピュータのメモリは有限であるため、この無限の要素すべてを直接的に保持するデータ構造を C++ で実装することは不可能である。しかし、コルーチン機能を利用することで、ジェネレータという形で自然数の概念を表現することができる。

```
generator<int> natural_number() {
    int n = 1;
    while(1){
        co_yield n;
        n++;
    }
}
```

例えば、このジェネレータ `natural_number` を使用して最初の 10 個を以下のように取得することができる。

```
int main() {
    for (int n : natural_number() | views::take(10)) {
        println("{}", n);
    }
}
```

2.1.7. 整数

2.1.6. と同様に考えて、整数全体をジェネレータを用いて表現する。0

```
generator<int> integer() {
    co_yield 0;
    int n = 1;
    while(1){
        co_yield n;
        co_yield -n;
        n++;
    }
}
```

最初の数個を取得するのは、2.1.6. と同様に行えばよい。

2.1.8. 偶数

2.1.7. で整数全体の集合を実装したため、その整数の集合の要素をそれぞれ 2 倍にすることで偶数の集合を得ることができる。

```
generator<int> even_number() {
    int n = 2;
```

```
    co_yield 0;
    while(1){
        co_yield n;
        co_yield -n;
        n+=2;
    }
}
```

2.1.9. 奇数

偶数のときと同様に考え、整数の集合を 2 倍して +1 することで奇数の集合を得ることができる。

```
generator<int> odd_number() {
    int n = 1;
    while(1){
        co_yield n;
        co_yield -n;
        n+=2;
    }
}
```

2.1.10. 有理数

有理数は以下のように構造体として実装することができる。分子と分母を任意の整数型で表現し、コンストラクタで自動的に既約分数に正規化される。また、比較演算子や四則演算もサポートしている。 `assert` を用いて分母が0でないことを保証し、数学的に正しい有理数のみを扱うようになっている。`normalize`関数は有理数を標準形に変換する。まず分母が負の場合は分母の符号を正にし、その後`gcd`（最大公約数）を用いて分子と分母を最大公約数で割ることで既約分数にする。これにより、異なる表現の同じ有理数（例：2/4と1/2）が同一の形式で保存される。 四則演算は

```
template<typename T>
struct Rational {
    T numerator;
    T denominator;

    Rational(T num = 0, T den = 1) : numerator(num), denominator(den) {
        assert(denominator != 0);  // 分母が0でないことを保証
        normalize();
    }
    void normalize() {
        if (denominator < 0) {
            numerator = -numerator;
            denominator = -denominator;
        }
        T g = gcd(abs(numerator), denominator);
        numerator /= g;
        denominator /= g;
    }
```

```
    void print() const {
        if (denominator == 1) {
            println("{}", numerator);
        } else {
            println("{}/{}", numerator, denominator);
        }
    }

    auto operator<=>(const Rational<T>& other) const {
        return (this->numerator * other.denominator) <=> (this->denominator *
other.numerator);
    }

    Rational operator+ (const Rational<T>& other) const {
        return { this->numerator * other.denominator + this->denominator *
other.numerator, this->denominator * other.denominator };
    }
    Rational operator- (const Rational<T>& other) const {
        return { this->numerator * other.denominator - this->denominator *
other.numerator, this->denominator * other.denominator };
    }
    Rational operator*(const Rational<T>& other) const {
        return { this->numerator * other.numerator, this->denominator *
other.denominator };
    }
    Rational operator/(const Rational<T>& other) const {
        return { this->numerator * other.denominator, this->denominator *
other.numerator};
    }
};
```

以下では大小比較と四則演算の例を示す。

```
int main() {
    Rational<int> a(-1, 2);
    Rational<int> b(2, 3);
    Rational<int> c(-4, -6);

    if(b == c) {
        b.print();
        printf("=");
        c.print();
    }else {
        b.print();
        printf("!=");
        c.print();
    }
    //output b = c

    (a + b).print();
    //output 1/6
```

```
    (a - b).print();
    //output -7/6

    (a * b).print();
    //output -1/3

    (a / b).print();
    //output -3/4
}
```

2.1.11. 有理数の集合

カントールの対角線論法を用いることですべての有理数を重複なく列挙できる。`gcd(num, den) == 1` の時は既約分数であるから、取得している。絶対値が同じ有理数を同時に生成することで、有理数全体を列挙できるようにした。

```
template<typename T>
generator<Rational<T>> rational_numbers() {
    co_yield Rational<T>(0, 1);
    T num = 1;
    T den = 1;
    while(1) {
        if(gcd(num,den) == 1) {
            co_yield Rational<T>(num, den);
            co_yield Rational<T>(-num, den);
        }
        num++;
        den--;
        if(den == 0) {
            den = num;
            num = 1;
        }
    }
}
```

以下のように行うことで、最初の数個を取得することができる。

```
int main() {
    for (const Rational<int>& r : rational_numbers<int>() | views::take(10)) {
        r.print();
    }
}
//output 0 1 -1 1/2 -1/2
```

2.1.12. 実数

実数全体の集合を実装することは不可能である。しかし、近似はできる。

2.1.13. ブール集合

$\mathbb{B}$ を表すには、`set` の要素を `bool` 型にすればよい。

```
set<bool> mySet = {true, false, true, false, true};
println("mySet is: {}", mySet);
//output mySet is: {false, true}
```

2.1.14. 属する

C++ では属するかどうかは `std::set` のメンバ関数 `find()` を用いることで確認することができる。`find()` は見つかった要素へのイテレータを返す。そうでない場合は、 `end()` を返す。

```
set<int> mySet = {1, 2, 3, 4};
decltype(mySet)::iterator it_1 = mySet.find(1);
if (it_1 != mySet.end()) {
    printf("%s\n", "1 is in the set");
} else {
    printf("%s\n", "1 is not in the set");
}
//output 1 is in the set

decltype(mySet)::iterator it_5 = mySet.find(5);
if (it_5 != mySet.end()) {
    printf("%s\n", "5 is in the set");
} else {
    printf("%s\n", "5 is not in the set");
}
//output 5 is not in the set
```

2.1.15. 外延的定義

外延的定義とは、集合の要素を明示的に列挙することで集合を定義する方法である。C++では`std::set`の初期化子リストを用いて外延的定義を直接実装できる。

```
set<int> mySet = {1, 2, 3, 4, 5};
println("mySet is: {}", mySet);
//output mySet is: {1, 2, 3, 4, 5}
```

2.1.16. 内包的定義

$\mathbb{N} = {1, 2, 3, \ldots}$ のように C++ で表すには、2.1.11. と同様に `generator` を用いればよい。

2.1.17. 多重集合

外延的定義において、ある要素が現れる回数が異なるとき異なる集合と考える場合の集合を多重集合という。C++では`multiset`を用いて実装することができる。

```
multiset<int> mySet = {0, 1, 2, 2, 3, 3, 4, 5};
println("mySet is: {}", mySet);
//output mySet is: {0, 1, 2, 2, 3, 3, 4, 5}
```

2.1.18. 集合族

集合族とは、集合を要素とする集合のことである。C++では`set<set<型>>`を用いて実装することができる。

```
int main() {
    set<set<int>> family = {{1, 2, 3}, {4, 5}, {1, 3, 5}, {4, 5}};
    println("family is: {}", family);
    //output family is: {{1, 2, 3}, {1, 3, 5}, {4, 5}}
}
```

この例では、`{4, 5}`が重複しているため、最終的には 3 つの異なる集合のみが集合族に含まれる。集合族の要素数や特定の集合が含まれているかの確認も通常の集合操作と同様に行える。

```
int main() {
    set<set<int>> family = {{1, 2}, {3, 4}, {1, 2, 3}};

    printf("family size is : %d\n", family.size());
    //output family size is 3

    if(family.find({1, 2, 3}) != family.end()) {
        printf("{1, 2, 3} is in family\n");
    }
    else {
        printf("{1, 2, 3} is not in family \n");
    }
    //output {1, 2, 3} is in family
}
```

2.1.19. 位数

2.1.20. 部分集合

部分集合の数

2.1.21. 集合の同一性

2.1.22. 真部分集合

2.2. 組、列、記号列

2.2.1. 組

2.2.2. 成分

2.2.3. 大きさ

2.2.4. 順序対

2.2.5. 組の同一性

2.2.6. 列

列の長さ

2.2.7. 記号列

2.2.8. 空列

2.3. 集合演算

2.3.1. 共通部分集合・積集合

2.3.2. 互いに素な集合

2.3.3. 和集合

2.3.4. 補集合

2.3.5. 差集合

2.3.6. 直積集合

2.3.7. 直和集合

2.3.8. べき集合

要素数、位数

2.3.9. 法則の確認

冪等則、交換則、結合則、分配則、吸収則、ド・モルガン

2.4. 写像

2.4.1. 写像、定義域、値域

map

2.4.2. 多変数関数

2.4.3. ブール関数

2.4.4. 二項演算

中置記法

2.4.5. 写像の同一性

2.4.6. 写像の集合

2.4.7. 写像の合成

2.4.8. 写像の結合則

2.4.9. 単射

2.4.10. 全射

2.4.11. 全単射

2.4.12. 恒等写像

2.4.13. 逆写像

2.4.14. 特性関数

2.4.15. 置換

2.4.16. 写像のグラフ

0 か 1 かを返せばよい

2.5. 関係

2.5.1. 関係

2.5.2. 関係の同一性

2.5.3. 関係の合成

2.5.4. 合成関係の結合則

2.5.5. 関係のべき乗

たどるのをかけばいいだけ

2.5.6. 反射律

2.5.7. 対称律

2.5.8. 反対称律

2.5.9. 推移律

2.5.10. 閉包

2.5.11. 同値関係

2.5.12. 同値類

2.5.13. 商集合

2.5.14. 半順序関係

2.5.15. 全順序関係

3. アピールポイント