Тема 10 Обобщени алгоритми

4.3. Обобщени алгоритми

4.3.1 Създаване на обобщени алгоритми

4.3.2 Алгоритми, предоставяни от библиотеката STL:

- ***Сменящи последователността операции;***
- ***Несменящи последователността операции ;***
- ***Сортиращи и релационни операции;***
- ***Обобщени числови операции.***

4.3. Обобщени алгоритми

Дефиниция:

Обобщените (generic) алгоритми са типизирани функции, предназначени за обработка (изпълняване на операции) върху данните, които се съхраняват в контейнерите.

Правила за създаване:

- Параметризирани са по типовете итератори;
- Използват шаблони за съответстващите структури данни (класовете) с които оперират;
- Независимо от организацията на данните в контейнера, той се разглежда като последователност от елементи.

4.3.1 Създаване на обобщени алгоритми

Пример: Създаване на алгоритъм binary_search:

Преобразува се алгоритъм за двоично търсене от конвенционален в обобщен (generic).

Дадена е функция, реализирана на C (C++) с параметри:

- Масив от тип int, предаден чрез указател;
- Брой на елементите в масива;
- Търсената стойност.

Връщаната стойност е константен указател към елемента ако е намерен или указател NULL в противен случай.

4.3.1 Създаване на обобщени алгоритми

Начална програма (глобална функция):

```
const int* binary_search (const int* array, int n, int x) {
   const int *lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return NULL;
}
```

4.3.1 Създаване на обобщени алгоритми

Тестова програма:

```
int main(){
  int test[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  const int* res = binary_search(test, 10, 5); // 5=>11
  if (res != NULL){
    cout << "found!\n";
  }
  else{
    cout << "no found!\n";
  }
  return 0;
}
```

4.3.1 Създаване на обобщени алгоритми

Обобщаване на типовете данни-Преобразуване на функцията binary_search в шаблонна функция.

```
template<class T>
const T* binary_search (const T* array, int n, const T& x) {
   const T *lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return NULL;
}
```

4.3.1 Създаване на обобщени алгоритми

Преобразуване на връщаната стойност. В случай на неуспешно изпълнение обобщеният алгоритъм трябва да се върне указател, сочещ зад последния елемент от масива , т.е. array + n (указател зад края) вместо указателя NULL:

```
template<class T>
const T* binary_search (const T* array, int n, const T& x) {
   const T* lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (x == *mid) return mid;
      if (x < *mid) hi = mid; else lo = mid + 1;
   }
   return array + n;
}
```

Как се проверява резултата от действието? (if (res != NULL))

4.3.1 Създаване на обобщени алгоритми

В съответствие с концепциите на итераторите в STL като параметри на функцията вместо масива array (предаден чрез указател към първия му елемент) и размерността му n се специфицира указател към първия и след последния елементи :

```
template<class T>
const T* binary_search (T* first, T* last, const T& value) {
   const T *lo = array, *hi = array + n, *mid;
   while(lo != hi) {
      mid = lo + (hi - lo) / 2;
      if (value == *mid) return mid;
      if (value < *mid) hi = mid; else lo = mid + 1;
   }
   return last;
}
```

4.3.1 Създаване на обобщени алгоритми

Замяна на указателите с итератори. Алгоритъмът има изчисления с указатели, поради което изисква итератори с произволен достъп:

```
template<class RandomAccessIterator, class T>
RandomAccessIterator binary_search(RandomAccessIterator
    first, RandomAccessIterator last,       const T& value) {
  RandomAccessIterator not_found = last;
  RandomAccessIterator mid;
  while(first != last) {
    mid = first + (last - first) / 2;
    if (value == *mid)
      return mid;
    if (value < *mid)
      last = mid;
    else
      first = mid + 1;
  }
  return not_found;
}
```

4.3.1 Създаване на обобщени алгоритми

Пример за използване с прости типове данни:

```
int main(){
 int test[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int search = 5;
  const int* res = binary_search( &test[0], &test[10], search);
  if (res != &test[10]){
    cout << "found!\n";
  }
  else{
    cout << "no found!\n";
  }
  return 0;
}
```

4.3.2 Алгоритми, предоставяни от библиотеката STL Сменящи последователността операции

- ***Сменящи последователността операции;***

  Предназначение:

  - осигуряват различни стратегии за изтриване или промяна на елемент (последователност от елементи) от редицата:

copy(), reverse_copy(), iter_swap(), remove(), remove_copy(), remove_if(), remove_copy_if(), replace(), replace_copy(), replace_if(), replace_copy_if(), swap(), swap_range(), unique(), unique_copy()

Добавени от C++11:

copy_if, copy_n, move, move_backward

Сменящи последователността операции

Принципи при създаване на модификатори:

- **if**
- **copy**

Пример:

void reverse(BidirectionalIterator first, BidirectionalIterator last);

и

OutputIterator reverse_**copy**(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);

които представят обединяване на операции.

reverse_copy изпълнява операцията присвояване:

*(result + (last - first) - i) = *(first + i)

За всяко i от интервала 0 <= i < (last - first)

Сменящи последователността операции

Пример: replace**_copy_if**:

Декларация на алгоритъма е следната:

```
template <class Iterator,class OutputIterator,
      class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
    OutputIterator result, Predicate pred,      const T&
    new_value);
```

Сменящи последователността операции

Примерна програма за използване на replace_copy_if:

```
const int MAX_ELEMENTS = 8 ;
 // Дефиниране на тип вектор от клас int
  typedef vector<int > IntVector ;
//Дефиниране на тип iterator към вектор от класа
  typedef IntVector::iterator IntVectorIt ;
//входен и изходен вектори
 IntVector Numbers(MAX_ELEMENTS);
 IntVector Result(MAX_ELEMENTS) ;
// работни итератори
  IntVectorIt startInt,endInt,itInt,lastInt,resultItInt;
// запълване на вектора с примерни данни
Numbers[0] = 10 ;   Numbers[1] = 20 ;
Numbers[2] = 10 ;  Numbers[3] = 15 ;
Numbers[4] = 12 ;  Numbers[5] = 7 ;
Numbers[6] = 9 ;  Numbers[7] = 10 ;
```

Сменящи последователността операции

```
// първи елемент в Numbers
  startInt = Numbers.begin() ;
//зад последния елемент в Numbers
  endInt = Numbers.end() ;
// първи елемент в Result
  resultItInt = Result.begin() ;
// извеждане на Numbers в изхода
  cout << "Numbers { " ;
  for(itInt = startInt; itInt != endInt; itInt++)
    cout << *itInt << " " ;
  cout << " }\n" << endl ;
// копиране на елементите от Numbers в Result,
// заменяйки тези, които са по-големи от 10 с 30
  lastInt = replace_copy_if(startInt, endInt,
     resultItInt, bind2nd(greater<int>(), 10), 30) ;
```

Сменящи последователността операции

```
// първи елемент в Result
  startInt = Result.begin() ;
//  последен елемент в Result
  endInt = Result.end() ;
// извеждане на Result в изхода
  cout << "Result  { " ;
  for(itInt = startInt; itInt != endInt; itInt++)
    cout << *itInt << " " ;
  cout << " }\n" << endl ;
```

Изход: Numbers { 10 20 10 15 12 7 9 10 }
Result { 10 30 10 30 30 7 9 10 }

bind2nd - Помощната темплейтна функция, която създава адаптер за преобразуване на бинарна обект функция в унарна обект функция чрез свързване на втория аргумент на бинарната функция с подадената стойност.

Сменящи последователността операции

Дефиниция и значение на параметрите:

```
template<class Operation, class Type>
   binder2nd <Operation> bind2nd(
      const Operation& _Func,
      const Type& _Right
   );
```

_Func – Бинарната функция, подлежаща на преобразуване в унарна.
_Right – Стойност, с която бинарната функция се свързва.

Пример:

bind2nd(greater<int>(), 10)

Сменящи последователността операции C++11

Еквивалентен код, валиден по стандарта C++11:

```
vector<int > Numbers = { 10, 20, 10, 15, 12, 7, 9, 10 };
vector<int > Result(Numbers.size()); // вектор за резултата със същата
// извеждане на вектора с алгоритъма copy
cout << "Numbers : "<<endl;
copy(Numbers.begin(), Numbers.end(), ostream_iterator<int>(cout, " "));
cout << endl;
// копиране на елементите от Numbers в Result,
// заменяйки тези, които са по-големи от 10 с 30
auto lastInt = replace_copy_if(Numbers.begin(), Numbers.end(),
   Result.begin(), bind2nd(greater<int>(), 10), 30);
// извеждане на Result с алгоритъма copy
cout << "Result  : "<<endl;
copy(Result.begin(), Result.end(), ostream_iterator<int>(cout, " "));
cout <<endl;
```

Сменящи последователността операции

- **генериращи и променящи редицата алгоритми**

fill(), fill_n(), for_each(), generate(), generate_n(), transform()

Примери:

template <class InputIterator, class Function>

Function for_each (InputIterator first, InputIterator last, Function f);

Използване:

Изисква обект функция като аргумент. Предполага се, че f е константна функция чрез дереферирания итератор.

for_each прилага f към резултата от дереферирането на итераторите от границите [first, last) и връща f. Ако f връща стойност, тя се игнорира.

Запълване с val:

OutputIterator fill_n (OutputIterator first, Size n, const T& val)

Сменящи последователността операции

Пример: Сумиране чрез клас функция и алгоритъм for_each

```
template <class T>
class CSum {
  public:
// Незадължителен конструктор за инициализация на
//начална стойност на сумата
    CSum(const T& init_value) { m_sum = init_value; }
// Оператор функция, изпълняващ сумирането на стойност към
// сумата
   void operator() (const T& value) { m_sum += value; }
// Функция за получаване на сумата
   const T& read_sum() { return m_sum; }
  private:
// Статична променлива за съхраняване на сумата
    static T m_sum;
};
// Декларация на член променливата за цялочислен тип
int CSum<int>::m_sum;
// Декларация на член променливата за реален тип
double CSum<double>::m_sum;
```

Сменящи последователността операции

```
// Примерна главна програма
int main(void) {
// Създаване на опашка от 10  int със стойност 2
  deque<int> oIntDeque(10,2);
// Създаване на обект от класа сума и инициализация
  CSum<int> oFuncSumInt(0);
// Прилагане на алгоритъма върху опашката от int
  for_each(oIntDeque.begin(),oIntDeque.end(),oFuncSumInt );
// Изход на резултата
  cout << oFuncSumInt.read_sum()<<endl;
// Създаване на опашка от 3 реални числа със стойност 200.1
  deque<double> oDblDeque(3,200.1);
// Създаване на обект от класа сума и инициализация
  CSum<double> oFuncSumDbl(0.);
  for_each(oDblDeque.begin(), oDblDeque.end(), oFuncSumDbl );
  cout << oFuncSumDbl.read_sum()<<endl;
}
```

Изход: *20*

600.3

Сменящи последователността операции

а) OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperator op);

б) OutputIterator transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperator op);

а) Едноаргументната операция-операторът op се прилага върху всеки от елементите от редицата и получените стойности се записват в изходния итерационен интервал, започващ от result. Връщаната стойност е итератор зад последната генерирана стойност.

б) Двуаргументната операция op се прилага между елементите от двата итерационни интервала- [first1, last1) и [first2, last2) и получените стойности се записват в изходния итерационен интервал, започващ от result. Връщаната стойност е итератор зад последната генерирана стойност.

Сменящи последователността операции

Пример за transform с масив (с глобална функция):

```
int f(int val1, int val2) { return val2 / val1; }
{ int a1[5] = {2, 4, 8, 16, 32};
  int a2[5] = {2, 4, 4, 8, 16};
  ostream_iterator<int> out(cout, " ");
  transform(&a2[0], &a2[5], &a1[0], out, f);
}
```

От изпълнението и се получава изход:

а) 1 1 0.5 0.5 0.5 ; б) грешка ;

в) 1 1 0 0 0; г) 1 1 2 2 2 ;

```
transform(&a1[0], &a1[5], &a2[0], out, f);
```

Сменящи последователността операции

Пример за transform с vector:

```
int f(int val1, int val2) { return val2 / val1; }
{
  vector<int> a1{ 2, 4, 8, 16, 32 };
  vector<int>  a2{ 2, 4, 4, 8, 16 };
  ostream_iterator<int> out(cout, " ");
  transform(a2.begin(), a2.end(), a1.begin(), out, f);
  return 0;
}
```

Сменящи последователността операции

Пример за generate_n генериране на студентски данни (име и оценки)

```
class CStudentGen {
public:
  CStudentGen() { srand(time(0)); }
  CStudentData operator()() {
    static char c = 'a';
    double d=0, e=0;
    d = rand() % 4; d += 3;
    e = rand() % 4; e += 3;
    string name(&c);
    c++;
    return CStudentData(name, d, e );
  }
};
// използване
vector<CStudentData> vi;
  generate_n(back_inserter(vi), MAX_STUDENTS, CStudentGen());
```

Несменящи последователността операции

- **алгоритми за търсене 13 алгоритъма :**

  adjacent_find(), count(), count_if(), equal_range(),find(), find_end(), find_first_of(), find_if(), lower_bound(),upper_bound(), search(), search_n()

Реализирани с двоично търсене. Изисква сортирана редица!:

  binary_search() ,equal_range(), lower_bound() и upper_bound()

Допълнения от стандарта (C++11):

any_of – логическа, за проверка съществува ли елемент, удовлетворяващ функция;

none_of – липса на такъв;

all_of – следва пример;

is_permutation – логическа проверка за две граници относно пермутация

Несменящи последователността операции

Примери за декларации на find и find_if :

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,  const T& value);
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

Пример за дефиниция на all_of C++11

```
template<class InputIterator, class UnaryPredicate> bool all_of
    (InputIterator first, InputIterator last, UnaryPredicate pred) {
    while (first!=last) {
            if (!pred(*first)) return false; ++first; }
    return true;
}
```

Несменящи последователността операции

Пример за програмиране

```
// Декларация на предикатната клас функция
template <class T>
class CFindFirstGreaterThenMember {
    public:
// Конструктори
      CFindFirstGreaterThenMember() : x(0) {}
      CFindFirstGreaterThenMember(const T&  xx) : x(xx) {}
// Оператор, реализиращ предиката
      int operator() (const T& v) { return v > x; }
    private:
      T x;

};
```

Несменящи последователността операции

```
// Тестова програма
vector<int> v={ 1, 2, 3, 4, 5}; // C++11
// Търсене на стойност, по-голяма от 4
vector<int>::iterator it=find_if(v.begin(),v.end(),
    CFindFirstGreaterThenMember<int> (4));
  it != v.end()? cout << *it << " " : cout << "not found";
// Търсене на стойност, по-голяма от 0 (инициализирана от
// подразбиращия се конструктор)
vector<int>::iterator it1=find_if (v.begin(), v.end(),
    CFindFirstGreaterThenMember<int> ());
  it1 != v.end()? cout << *it1 : cout << "not found";
```

Изход: *5 1*

C++11:

vector<int>::iterator ⇔ auto

Несменящи последователността операции

Някои алгоритми, като adjacent_find, получават бинарен предикат binary_pred от тип BinaryPredicate. Алгоритъмът adjacent_find връща първия итератор it, за който се изпълнява условието: binary_pred (*it, *(it+1)) == true.

Декларацията в стандартната библиотека може да се представи по следния начин:

```
template <class InputIterator, class BinaryPredicate>
InputIterator adjacent_find(InputIterator first, InputIterator last,
    BinaryPredicate binary_pred);
```

Несменящи последователността операции

Например-намиране на първата двойка от стойности, чиито произведение е нечетно с клас функция (предикат):

```
template <class T>
//Обект функция, определяща нечетност на произведението v1*v2
class CProdOdd {
    public:
    bool operator() (const T& v1, const T& v2)
      { return v1%2 != 0 && v2%2 != 0; }
};
// Тестова програма
list<int> lst;
// Запълване на списъка с 2 9 6 13 7
list<int>::iterator it3 = adjacent_find (lst.begin(), lst.end(),
                          CProdOdd <int>());
if(it3 != lst.end()) {
    cout << *it3 << " "; it3++; cout << *it3++; }
else
    cout << "not found";
```

Изход: *13 7*

Несменящи последователността операции

```
// Инициализация на deque C++11:

deque<string> player={"Pele", "Platini", "Stoychkov", "Stoychkov", "Zidane"};

// Намиране на двойка еднакви стрингове в опашката от имена:
auto => deque<string>::iterator  при C++11
  auto it4 = adjacent_find(player.begin(), player.end());
     cout << endl <<endl;
  if (it4 != player.end()) { cout << *it4++ << " "; cout << *it4++; }

// Намиране на двойка, в която първото име в опашката от имена
// е лексикографски  по-голямо от следващото:
  it4 = adjacent_find(player.begin(), player.end(), greater<string>());
     cout << endl <<endl;
  if (it4 != player.end()) { cout << *it4++ << " "; cout << *it4; }
     cout << endl <<endl;
```

Изход: *Stoychkov Stoychkov*
Platini Stoychkov

Несменящи последователността операции

1) ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value);
2) ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value, Compare comp);

Алгоритъмът работи върху подредени редици.

1)В случай, че елемент от редицата е не по-малък (т.е. по-голям или равен) на подадения елемент value, се връща итератора към този елемент. В противен случай се връща итератор зад последния елемент-last. Ако е намерен, това е първият итератор, по който може да се вмъкне value без да се наруши подредбата.

2)В случай, че предикатът comp, приложен върху елемент от редицата и подадения елемент value върне false, се връща итератора към този елемент. В противен случай се връща итератор зад последния елемент-last.

Сортиращи и релационни операции

Група 3 съдържа **сортиращи и помощни на сортирането алгоритми**

STL предоставя 14 базови сортиращи и подпомагащи сортирането алгоритми:

inplace_merge(), merge(), nth_element(), partial_sort(),

partial_sort_copy(), partition(), random_shuffle(), reverse(),

reverse_copy(), rotate(), rotate_copy(), sort(),

stable_sort(), stable_partition()

Допълнения от стандарта (C++11):

is_partitioned(), partition_copy(), partition_point()

is_sorted() , is_sorted_until(), shuffle ()

Сортиращи и релационни операции sort

Пример

```
vector<int> v={ 3, 1, 7, 5, 4, 2, 6};
sort (v.begin(), v.end() );
// Извеждане на съдържанието - copy
sort (v.begin(), v.end(), less<int>() );
// Извеждане на съдържанието - copy
sort (v.begin(), v.end(), greater<int>() );
// Извеждане на съдържанието
```

Изход: *1 2 3 4 5 6 7*

1 2 3 4 5 6 7

7 6 5 4 3 2 1

Сортиращи и релационни операции
nth_element

а) void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);

б)void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);

а) Всички елементи от итерационния интервал [first, last) се сортират по отношение на елемента, сочен от итератора nth като всички по-малки от него се разполагат в итерационния интервал [left, nth) а всички по-големи – в интервала [nth + 1, last). Двете редици са сортирани в съответствие с оператора (функция) operator<()

б) разликата е в условието, по което се сортират и сравняват елементите. За този прототип на алгоритъма то се определя по функцията (клас функцията) comp.

Сортиращи и релационни операции

random_shuffle, partition

random_shuffle(BidirectionalIterator first, BidirectionalIterator last)

За получаване на случаен ред в подредбата на елементите от редицата.

BidirectionalIterator partition(BidirectionalIterator first,

BidirectionalIterator last, UnaryPredicate pred);

Всички елементи от границата [first, last), за които едноместния предикат pred е true се поставят преди елементите, за които се върне false. Връщаната стойност е елемента след последния, за който предикатът е върнал true.

Обобщени числови операции

Числови алгоритми:

За да се използват трябва да се включи файл <numeric>.

accumulate(), partial_sum(), inner_product(), adjacent_difference()

1) Type accumulate(InputIterator first, InputIterator last, Type init);

2) Type accumulate(InputIterator first, InputIterator last, Type init, BinaryOperation op);

1) operator+()

2) в съответствие с op()

Върнатата от алгоритъма стойност е:

1) сума от елементите;

2) между елементите се прилага операцията BinaryOperation;

Обобщени числови операции

Числови алгоритми **accumulate():**

```
template <class InputIterator, class T>
T accumulate (InputIterator first, InputIterator last, T init) {
    while (first!=last) {
        init = init + *first; // или init=binary_op(init,*first)
        ++first;
    }
     return init;
}
```

Обобщени числови операции

- **partial_sum:**

1)OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);

2)OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);

В първия вариант на алгоритъма всеки елемент на резултантния контейнер (освен първия) се получава като сума от всички предходни и текущия, а при втория вместо сума се прилага оператора (функция) op.

Обобщени числови операции

- **inner_product**

1)Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init);

2)Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init, BinaryOperator1 op1, BinaryOperator2 op2);

Резултатът се получава като с началната стойност init:

1) Се образува сума от почленните произведения на редиците от итерационния интервал;

2) Между всяка двойка елементи от редиците се прилага op1 (подразбира се умножение), а получените резултати, започвайки от init се свързват с op2 (подразбира се сумиране).

Обобщени числови операции

- **adjacent_difference**

1) OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);

2) OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);

Стойността на следващите елементи (след първия) е:

1) разлика между текущия и предходния;

2) между текущия Ei и предходния Ei-1 се прилага операцията op (Ei op Ei-1);

Множествени операции

Само за сортирани редици!

includes() – bool, ако множество е подмножество на друго

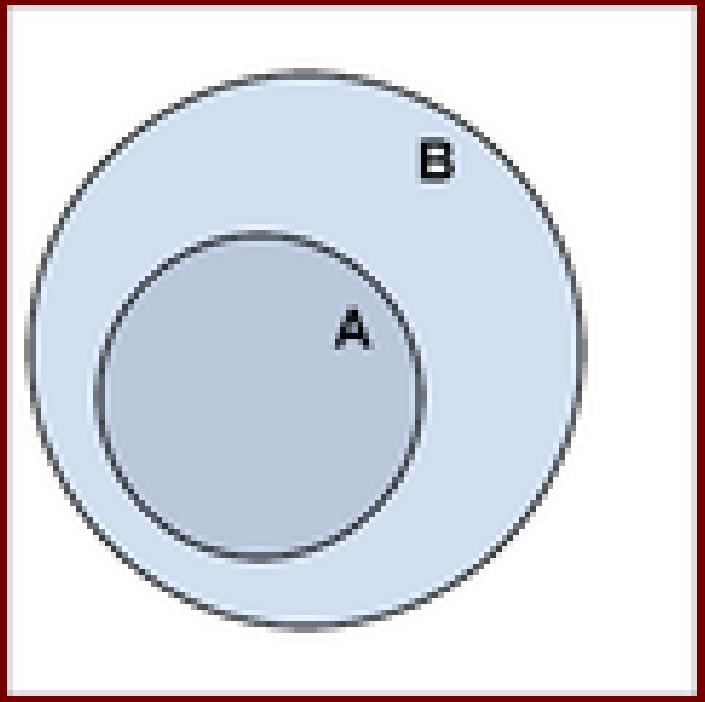
set_difference() : изчислява разлика на две множества в трето

set_intersection() : изчислява сечение на две множества в трето

set_symmetric_difference() : изчислява симетрична разлика на две множества в трето

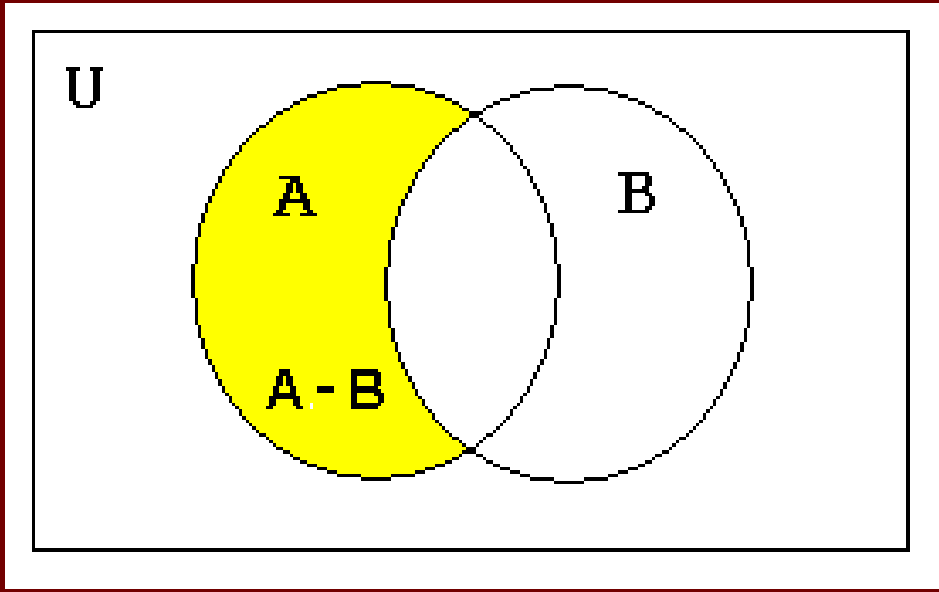
set_union() : изчислява обединение на на две множества в трето

Множествени операции : includes()



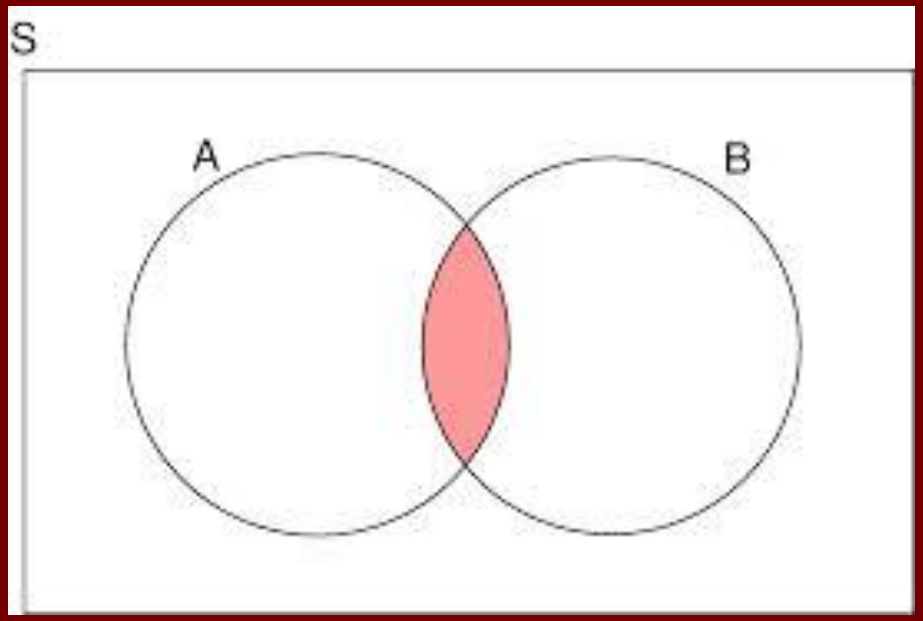
! Изисква предефиниране на operator <

Множествени операции : set_difference()



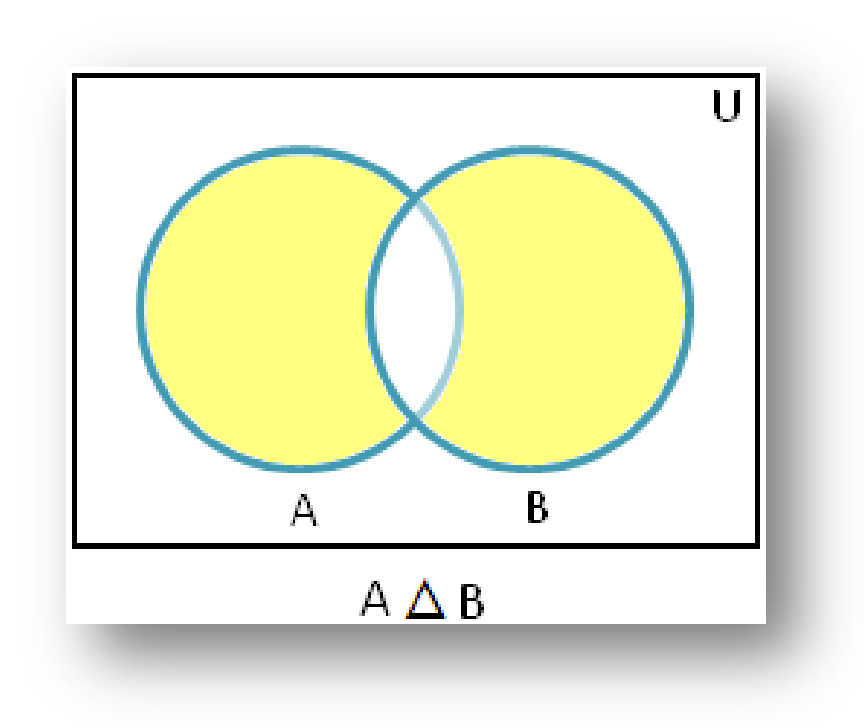
! Изисква предефиниране на operator <

Множествени операции : set_intersection()



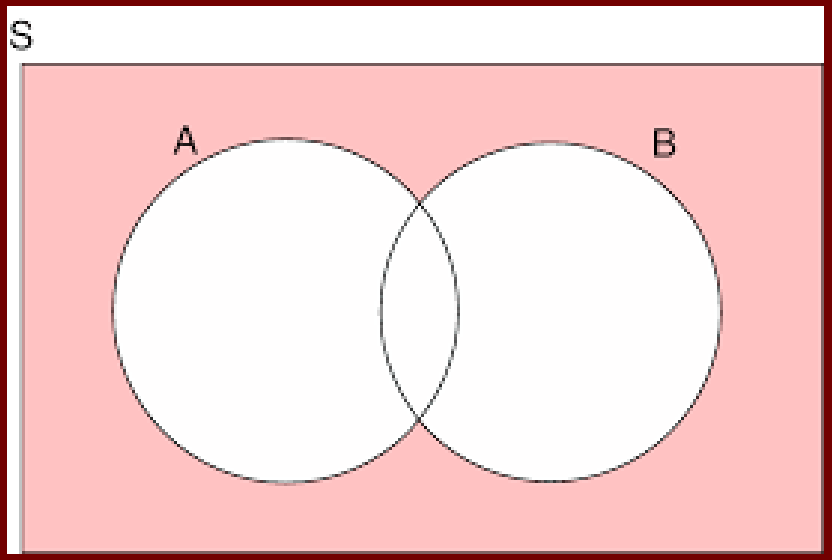
! Изисква предефиниране на operator <

Множествени операции : set_symmetric_difference()



! Изисква предефиниране на operator <

Множествени операции : set_union()



! Изисква предефиниране на operator <

Алгоритми – общи изисквания и ограничения към итераторите

	Input	Output	Forward	Bidirectional	Random Access
for_each	x				
find	x				
count	x				
copy	x	x			
replace			x		
unique			x		
reverse				x	
sort					x
nth_element					x
merge	x	x			
accumulate	x				

Обобщени алгоритми

Въпроси?