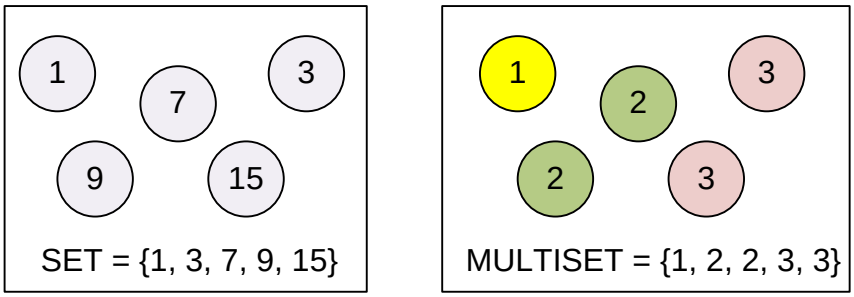
SET data structure

Set is a sorted associative container that can only store *unique* items.

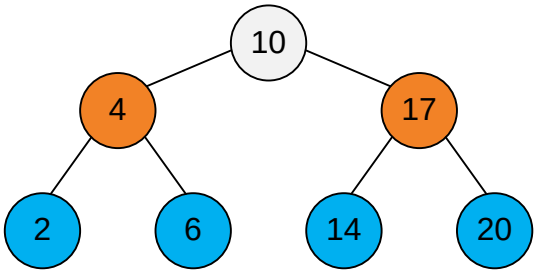
Multiset is an associative container, which, unlike a set, can contain duplicate elements.



To use a ***set*** or ***multiset*** in a program, include the <set> header file in it:

```
#include <set>
```

The internal structure of the set is a balanced binary tree, or rather a ***red – black tree***. Thus, access to its elements is performed with complexity O (log *n*), where *n* is the cardinality of the set.



The ***set*** container provides the following functionality:

- add an element to the set, excluding the possibility of duplicates;
- remove an element from the set;
- find out the number of (different) elements in the container;
- check if some element is present in the container.

Elements of a set and multiset belong to an arbitrary type T that supports assignment, copying, and comparison by sorting criteria. The default criteria is ***less***, which compares elements with operator <.

Declare a set of integers:

```
set<int> s;
```

Declare a multiset of integers:

```
multiset<int> s;
```

To insert an element into a set, use the ***insert*** method

```
s.insert(5);
```

The ***size*** method returns the number of elements in a set:

```
number_of_elements = s.size();
```

Example. Inserts the numbers 6 1, 5, 2, 6, 1 into the set. If the inserted element already exists in the set, then it is ignored (not added again). Thus, we get an ordered set {1, 2, 5, 6}, consisting of 4 elements.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;

int main(void)
{
  s.insert(6); s.insert(1); s.insert(5);
  s.insert(2); s.insert(6); s.insert(1);
  printf("%d\n",s.size());
  return 0;
}
```

E-OLYMP 5337. Different - different *n* integers are given. How many of them are different?

► Insert all numbers into the set `set<int>`. Print the size of the set.

Example. Set of strings can be also used.

```
#include <cstdio>
#include <set>
#include <string>
using namespace std;

set<string> s;

int main(void)
{
  s.insert("Baku");   s.insert("Kiev");
  s.insert("Moscow"); s.insert("Tbilisi");
  printf("%d\n", s.size());
  return 0;
}
```

E-OLYMP 7546. I've been Everywhere, Man Each test contains a list of the cities Alice has visited. How many different cities did she visit?

► Insert all cities into the set `set<string>`. Print the size of the set.

An ***iterator*** is an interface that provides access to and navigation through the elements of a collection (array or container). In the simplest case, an iterator is a pointer (a variable containing a memory address).

The iterator *iter* of a set is declared as follows:

```
set<int>::iterator iter;
```

Only two operations can be performed over the iterators: moving forward (*iter*++) – moving to the next element of the set and backward (*iter*--) – moving to the previous element.

The ***begin*** method returns the address of the first (smallest) element in the set. One can assign only addresses to iterators.

```
iter = s.begin();
```

If the iterator *iter* contains the address of an element of the set, then the element itself is available as **iter*. For example, the first element of a set can be printed as follows:

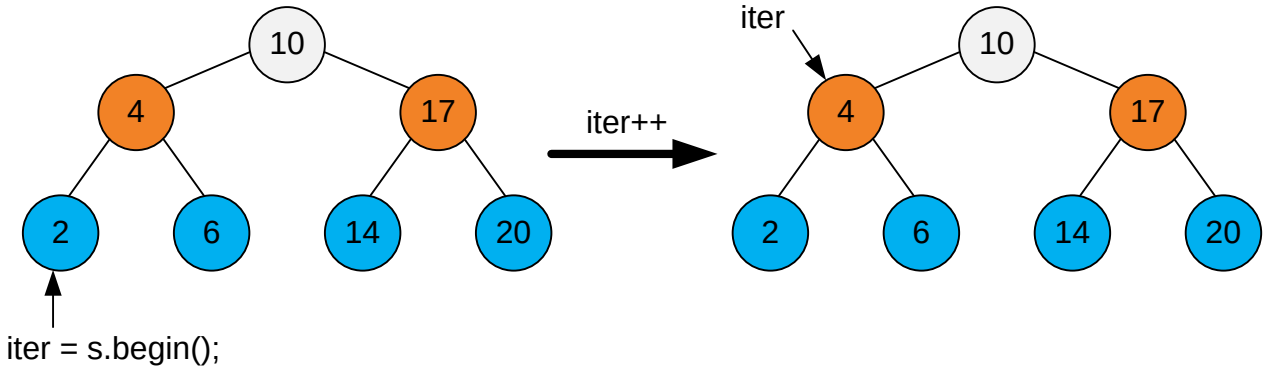
```
iter = s.begin();
printf("%d\n",*iter);
```

The first element of the set can be printed as follows:

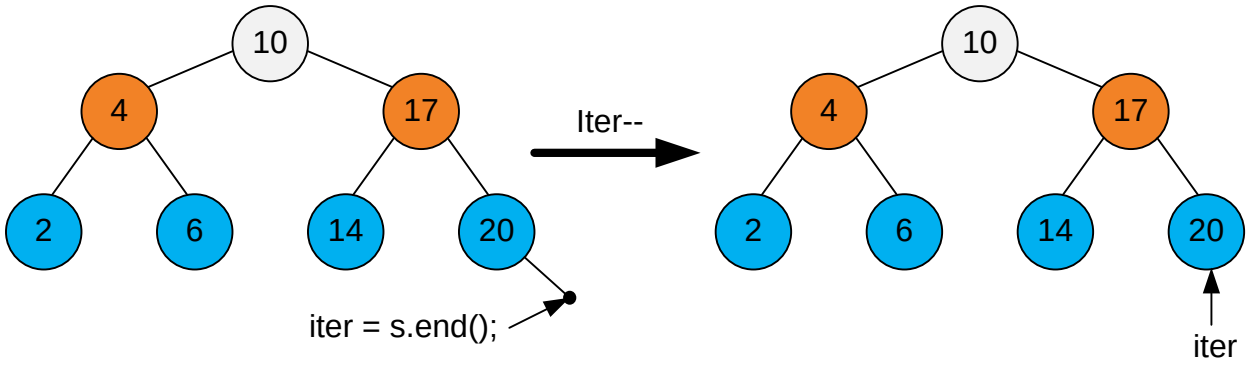
```
printf("%d\n",*s.begin());
```

Example. Print the first and the second element of the set.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;
set<int>::iterator iter;

int main(void)
{
  s.insert(10); s.insert(4); s.insert(17);
  s.insert(2);  s.insert(6); s.insert(14); s.insert(20);

  iter = s.begin();  printf("%d\n",*iter); // first element
  iter++;            printf("%d\n",*iter); // second element
  return 0;
}
```



The ***end*** method returns the address behind the last element. It does not point to any element, so the *s.end() operation is meaningless. If the set is empty, then s.begin() = s.end().



Example. In order to print the *last* (*largest*) element of the set, it is necessary to set the iterator *iter* to s.end(), decrease it by 1 (decrement is possible only through the *iter*-- operation) and print the value it points to.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;
set<int>::iterator iter;

int main(void)
{
  s.insert(10); s.insert(4); s.insert(17);
  s.insert(2);  s.insert(6); s.insert(14); s.insert(20);

  iter = s.end(); iter--;
  printf("%d\n",*iter); // last element
  return 0;
}
```

You can iterate over the elements of the ***set*** using the following loop (set the iterator *iter* to the first element, and then increment it by one using the ++ operator until it reaches s.end()):

```
for(iter = s.begin(); iter != s.end(); iter++)
 printf("%d ",*iter);
```

E-OLYMP 4848. Quick sort Sort the given sequence in non-decreasing order.

► Since numbers can be repeated in a sequence, use `multiset<int>`. The amount of input numbers is not known, read them till the end of file and insert them into multiset.

Print the numbers of the multiset in non-decreasing order.

```
#include <cstdio>
#include <set>
using namespace std;

int x;
multiset<int> s;
```

```
multiset<int>::iterator iter;

int main(void)
{
  while (scanf("%d", &x) == 1)
    s.insert(x);

  for (iter = s.begin(); iter != s.end(); iter++)
    printf("%d ", *iter);
  printf("\n");
  return 0;
}
```

Example. Print the words in alphabetical order.

```
#include <cstdio>
#include <set>
#include <string>
using namespace std;

set<string> s;
set<string>::iterator iter;

int main(void)
{
  s.insert("Baku");   s.insert("Kiev");
  s.insert("Moscow"); s.insert("Tbilisi");
  for (iter = s.begin(); iter != s.end(); iter++)
    printf("%s\n", (*iter).c_str());
  return 0;
}
```

E-OLYMP **5089. Vocabulary** There is a set of words for the orcs. Print an orc dictionary (a list of words in alphabetical order).

► Insert all words into a set `set<string>`. Print the words in alphabetical order.

Using the ***copy constructor***, one can insert elements from array into a set. The initialization of a set when it is declared looks as follows:

```
int m[10] = {3, 4, 10, 1, 9, 13, 2, 8, 20, 17};
set<int> s(m,m+10);
```

One can copy numbers from an array to a set as follows:

```
s = set<int>(m,m+10);
```

Example. An array of integers is given. Remove the duplicate elements from it and print the remaining ones in a sorted form.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;
set<int>::iterator iter;
int m[10] = {3, 4, 1, 1, 9, 2, 2, 17, 20, 17};
```

```
int main(void)
{
  s = set<int>(m,m+10);
  for(iter = s.begin(); iter != s.end(); iter++)
   printf("%d ",*iter);
  printf("\n");
  return 0;
}
```

Example. Compute the sum of the set elements.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;
set<int>::iterator iter;
int sum;

int main(void)
{
  s.insert(10); s.insert(4); s.insert(17);
  s.insert(2);  s.insert(6); s.insert(14); s.insert(20);

  sum = 0;
  for(iter = s.begin(); iter != s.end(); iter++)
   sum = sum + *iter;

  printf("%d\n",sum);
  return 0;
}
```

You can remove the elements from a set using the ***erase*** method. You can delete by value (all values will be deleted from the multiset):

```
s.erase(4);
```

You can pass an iterator to the ***erase*** method. Then the element it points to will be removed:

```
s.erase(iter);
```

For example, you can remove the first element of a set like this:

```
s.erase(s.begin());
```

Removing the last element:

```
iter = s.end(); iter--; s.erase(iter);
```

After the operation s.erase(*iter*), the iterator *iter* points to nothing and cannot be used with any further increment or decrement operations. However, after the s.erase (*iter*++) operation, the iterator will point to the item next to the item being removed. Accordingly, after s.erase(*iter*--), the iterator will point to the element before the deleted one.

Example. Removing from a set all even elements.

```
#include <cstdio>
#include <set>
```

```
using namespace std;

set<int> s;
set<int>::iterator iter;

int main(void)
{
  s.insert(10); s.insert(4); s.insert(17);
  s.insert(2);  s.insert(1); s.insert(23);

// after operation s.erase(iter) the increment iter++ gives RTE
//for(iter = s.begin(); iter != s.end(); iter++)
//  if (*iter % 2 == 0) s.erase(iter);

  for(iter = s.begin(); iter != s.end();)
    if (*iter % 2 == 0) s.erase(iter++); else iter++;

  for(iter = s.begin(); iter != s.end(); iter++)
    printf("%d ",*iter);
  printf("\n");
  return 0;
}
```

Find the element. The ***find***(*x*) method searches the number *x* in the set and returns an iterator. If *x* is found, then the iterator points to it. Otherwise, the iterator points to s.end().

Example. Find the element *x* = 17.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;
set<int>::iterator iter;

int main(void)
{
  s.insert(10); s.insert(4); s.insert(17);
  s.insert(2);  s.insert(1); s.insert(23);

  iter = s.find(17);

  if (iter != s.end())
    printf("Found %d\n",*iter);
  else
    printf("Not found\n");

  return 0;
}
```

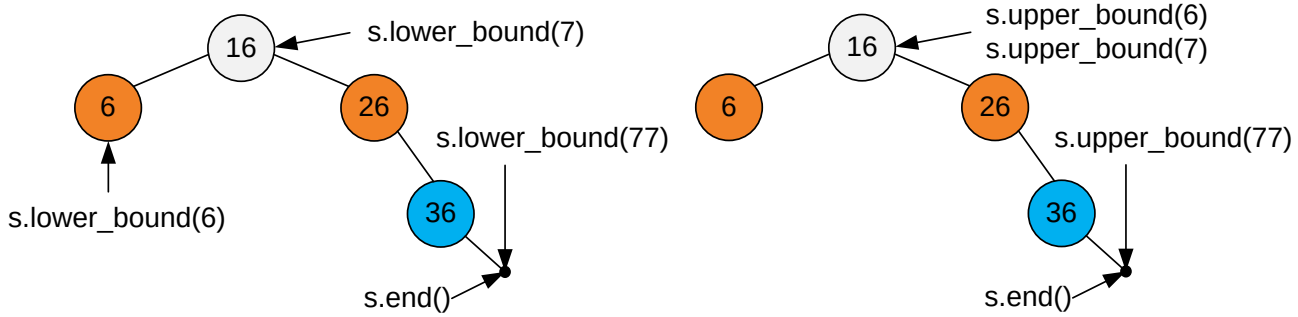
lower_bound. The ***lower_bound***(*key*) function searches the set for the key *key*. It returns:

- an iterator to an element *key* in the set, if *key* is present in it;

- an iterator to the next element greater than *key*, present in the set, if *key* is not present in it;
- an iterator to s.end() if *key* is greater than the maximum element of the set;

Summarizing the above, we can say that ***lower_bound***(*key*) returns an iterator to the first element that is not less than *key*.

```
#include <cstdio>
#include <set>
using namespace std;

set<int> s;
set<int>::iterator iter;

int main(void)
{
  s.insert(6); s.insert(16);
  s.insert(36); s.insert(26);

  // 6 is present
  iter = s.lower_bound(6);
  printf("%d\n", *iter);

  // 7 is NOT present
  // points to next greater after 7
  iter = s.lower_bound(7);
  printf("%d\n", *iter);

  // 777 exceeds the max element in set
  iter = s.lower_bound(77);
  if (iter == s.end()) puts("Greater than max element");
  else printf("%d\n", *iter);

  return 0;
}
```



upper_bound. The function ***upper_bound***(*key*) returns an iterator to the first element strictly greater than *key*.

Works the same as ***lower_bound*** except when *key* is present in the set. In this case, ***lower_bound*** returns an iterator to *key*, and ***upper_bound*** returns an iterator to the next element after the *key*.

Set of pairs. When creating multiple pairs, they will be sorted by the first component. The pair is created with the ***make_pair*** function.

```
#include <cstdio>
#include <set>
using namespace std;

set<pair<int,int> > s;
set<pair<int,int> >::iterator iter;
int i;

int main(void)
{
  for(i = 1; i < 10; i ++) s.insert(make_pair(100 - i*i,i));

  for(iter = s.begin(); iter != s.end(); iter++)
   printf("%d %d\n",(*iter).first, (*iter).second);
  return 0;
}
```

Comparator. To compare the stored elements with each other, the ***set*** container refers to the ***comparator***.

E-OLYMP **8637. Sort the points** The coordinates of *n* points are given on a plane. Print them in increasing order of sum of coordinates. In the case of equal sum of point coordinates sort the points in increasing order of abscissa.

► Create a **Point** class. When inserting points into the ***multiset*** *s* (input can contain equal points), sort them by the abscissa *x*, and if they are equal, by the ordinate *y*. The sorting function is defined as the comparator < on the **Point** class.

```
#include <cstdio>
#include <set>
using namespace std;

int i, x, y;

struct Point
{
public:
  int x, y;
  Point(int x, int y) : x(x), y(y) {}
};

int operator< (Point a, Point b)
{
  if (a.x + a.y == b.x + b.y) return a.x < b.x;
  return a.x + a.y < b.x + b.y;
}

multiset<Point> p;
multiset<Point>::iterator iter;

int main(void)
{
  while (scanf("%d %d", &x, &y) == 2)
    p.insert(Point(x, y));
```

```
  for (iter = p.begin(); iter != p.end(); iter++)
    printf("%d %d\n", (*iter).x, (*iter).y);

  return 0;
}
```