Lecture 4 ADT Set, Map, Dictionary. Iterators

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms* September 12th, 2024

IDA, Linköping University

3

Content

Contents

Symbol table						
1.1	Abstract datatype					
1.2	Associative container i C++					

2 Iterators

1 Symbol table

Symbol table

- Abstraction of key-value pairs
 - *Submit* a value with the specified key
 - Given a key, search by the corresponding values

1.1 Abstract datatype

ADT Set

- Storage of keys
- Typical operations:
 - size() the number of keys
 - isEmpty() returns true if there are no keys
 - contains(k) returns **true** if k is used in the container, otherwise **false**
 - put(k) add k to the container
 - remove(k) remove k from the container

ADT Map

- Storage of pairs (*key*, *value*) with at most one pair per key. More than one key may map to the same value!
- Typical operations:
 - size() number of pairs in the map
 - isEmpty() checks if the map is empty
 - get(k) returns the value associated with k or **null** if the key is not in the map
 - put(k,v) add (k,v) to the container and returns **null** if k is new; otherwise replace the value with v and returns the old value
 - remove(k) remove the pair (k, v) and return v; return **null** if the key is not in the map

4.5

4.1

4.2

4.3

ADT Dictionary

- Storage of pairs (*key*, *value*), possibly several pairs per key Contains the relationship between a key and one or several values!
- Typical operations:
 - size() number of elements
 - isEmpty() check if the container is empty
 - find(k) returns any item with k or **null** if there are none
 - findAll(k) return a list of value with key k
 - insert(k, v) add (k, v) and return the new entry
 - remove(k, v) remove (k, v) and returns the value; return **null** if there are no matching pair
 - entries() returns a list of all entries

1.2 Associative container i C++

pair class

Class for storing pairs - for example, used by all map-containers.

• has a helper function make_pair to create a pair object

```
vector< pair<int, string> > v;
int i = 4711;
string s = "foobar";
v.push_back(make_pair(i, s)); // exactly like pair<int, string>(i, s);
```

• it is easy to access the pair content:

```
pair<int, string> myPair;
myPair.first = 4711;
myPair.second = "foobar";
```

Associative container

Quick access to data based on key search

• In the following associative containers keys are in order — iterating over them follow the ordering of the key

map multimap set	multiset
------------------	----------

- · set-containers stores only a key
- map-containers store key-values pairs using pair to store a key and the associated value
- non-multi variant allows for a unique key
- · Moreover, there are corresponding unordered associative container:

unordered_map unordered_multimap unordered_set unordered_multiset

Associative containers - operations

Storlek, kapacitet	map	multimap	set	multiset
n = a.size()	•	•	•	•
<pre>n = a.max_size()</pre>	•	•	•	•
<pre>b = a.empty()</pre>	•	•	•	•

Jämförelse	map	multimap	set	multiset
== !=	•	•	•	•
< <= > >=	•	•	•	•

Elementåtkomst	map	multimap	set	multiset
x = a[k]	•			
x = a.at(k)	•			

Anm: om ett element med nyckel k inte finns i en map, skapas det av **operator**[], med det associerade värdet defaultinitierat. En referens till det associerade värdet returneras och kan användas för att tilldela ett värde.

4.7

4.6

4.8

Associative containers — operations

Modifierare	map	multimap	set	multiset
<pre>pair<iterator, bool=""> p = a.insert(x);</iterator,></pre>	•		•	
<pre>it = a.insert(x)</pre>		•		•
it = a.insert(pos, x)	•	•	•	•
a.insert(first, last)	•	•	•	•
a.insert({ x, y, z })	•	•	•	•
<pre>pair<iterator, bool=""> p = a.emplace(args);</iterator,></pre>	•		•	
it = a.emplace(args)		•		•
<pre>it = a.emplace_hint(pos, args)</pre>	•	•	•	•
a.erase(it)	•	•	•	•
n = a.erase(k)	•	•	•	•
a.erase(first, last)	•	•	•	•
al.swap(a2)	•	•	•	•
a.clear()	•	•	•	•
				1

1) i fallen map och multimap har x typen pair (nyckel-värde)

2) i fallen map och multimap ska *args* var värden motsvarande ett nyckel-värde-par 3) använd hellre **auto**

Associative containers - operations

Map- och set-operationer	map	multimap	set	multiset
<pre>it = a.find(k)</pre>	•	•	•	•
n = a.count(k)	•	•	•	•
<pre>it = a.lower_bound(k)</pre>	•	•	•	•
<pre>it = a.upper_bound(k)</pre>	•	•	•	•
<pre>pair<iter, iter=""> p = a.equal_range(k);</iter,></pre>	•	•	•	•

1

Specialiserad operation

swap(a1, a2)	•	•	•	•

1) använd hellre auto

Exercise - count unique word

• Write a program that counts the number of unique words in a large text file

Exercise - count words

• Write a program that determines which words are most common in a large text file

Exercise — anagram

• Write a program that computes the largest amount of anagrams from a dictionary

2 Iterators

Iterators

Can be seen as a pointer that can point to elements in a container and know how to move between elements

- Container iterator
 - point to an element in the container
 - each container class has its own iterator
- Stream iterator
 - bound to a stream
 - allows us to read from and write to streams using iterator operations
- insert iterator
 - bound to an iterator

4.10

4.11

4.12

4.13

- used to insert into the iterator
- move iterator
 - element values are moved from the source to the destination, instead of being copied source elements zeroed typically
- · past-the-end iterator
 - a special iterator value indicating the end of a container, a stream or other type of range of values
 - primarily used to compare with other iterators: "Have we reached the end?"

Iterators

- access to the element with * or -> (like a pointer)
- many operations on containers use iterators as arguments and/or return an iterator
- almost all algorithms use iterator for operation on containers and other datastructure, including streams
- improper use of an iterators can lead to execution errors, such as "segmentation fault"

Container iterator

Each type of container has its own specific iterator implementation:

- Containers have the following types:
 - iterator iterates from the beginning to the end of a container
 - const_iterator can be used to read but not modify
 - reverse_iterator iterates from the end to the beginning of a container
 - const_reverse_iterator
- Here are the functions to return iterators: begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend()
- Example:

```
for(vector<int>::const_iterator it = v.cbegin(); it != v.cend(); ++it)
{
    cout << *it << '_';
}</pre>
```

• If nothing should be changed during the operation, use the const version

Iterator position



• Reverse iterator vs normal iterator:

```
for (vector < int >:: const_reverse_iterator it = v.crbegin();
    it != v.crend(); ++it)
{
    cout << *it << '_';
}
// Equivalent to:
for (vector < int >:: const_iterator it = v.cend(); it != v.cbegin(); --it)
{
    cout << *(it-1) << '_';
}</pre>
```

4.16

4.17

Iterator category

An iterator "points" to a value. All operators supports ++it and it++ Different applications require different iterators. There is a hierarchy among iterator categories:



- InputIterator: can be referenced to read value.
- OutputIterator: can be referenced for writing a value.
- ForwardIterator: can both read and write.
- BidirectionalIterator: can move back and forth.
- RandomAccessIterator: allow pointer arithmetic and arbitrary index.

Operations on iterators

Förutom kopiering och tilldelning finns följande operationer för de olika iteratorkategorierna:

	Input	Output	Forward	Bidirectional	Random Access
== !=	Ja	Ja	Ja	Ja	Ja
*	Läs	Skriv	Läs + Skriv	Läs + Skriv	Läs + Skriv
->	Läs	Skriv	Läs + Skriv	Läs + Skriv	Läs + Skriv
++	Ja	Ja	Ja	Ja	Ja
	-	-	-	Ja	Ja
+ +==	-	-	-	_	Ja
< <= > >=	-	-	-	_	Ja
it [n]	_	-	-	_	Ja
advance(<i>it</i> , <i>n</i>)	Ja	-	Ja	Ja	Ja
distance(<i>it1</i> , <i>it2</i>)	Ja	_	Ja	Ja	Ja
			forward_list de oordnade associativa containrarna	list set multiset map multimap	vector deque array

Example — iterator operations

Vissa iteratoroperationer nedan kräver random access-iteratorer.

<pre>int a[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };</pre>	// a är en pekare till första elementet		
<pre>vector<int> v1{ begin(a), end(a) };</int></pre>	<pre>// "range access"-funktioner (<iterator>)</iterator></pre>		
<pre>vector<int> v2{ v1.begin(), v1.end() };</int></pre>			
<pre>vector<int>::iterator first = v1.begin(); vector<int>::iterator past_end = v1.end();</int></int></pre>			
cout << *first << '\n';	// avreferering		
<pre>cout << *(past_end - 1) << '\n';</pre>	// aritmetik		
<pre>cout << past_end - first << '\n';</pre>	// avstånd		
cout << first[5] << '\n';	// indexering		
<pre>vector<int>::iterator it{ first };</int></pre>	// kopiering		
<pre>vector<int>::iterator middle = v2.begin() + (v2.end</int></pre>	() - v2.begin()) / 2; // tilldelning		
<pre>v2.erase(middle, v2.end());</pre>			

4.20