Popoveniuc Mircea
911

Sorted Map

Elements are key-value pairs. Keys are equality comparable (less-than comparable for sorted map). No duplicate keys are allowed. Order is defined by the less-than comparison on keys.

Domain

$D_{SM<TKey,\ TValue>} = S = \{s \mid s = \{(k_1,v_1), (k_2,v_2), \ldots, (k_n,v_n)\}, k_i \in TKey, v_i \in TValue, \forall i = 1,2, \ldots, n, k_1 < k_2 < \ldots < k_n\}$

Operations

constructor()

Data:	**Pre:**
Res: s	**Post:** $s \in S$ $s = \varnothing$ (empty sorted map)

copy_constructor(s)

Data: s	**Pre:** $s \in S$
Res: s’	**Post:** $s' \in S$ $s' = s$

destructor(s)

Data: s	**Pre:** $s \in S$
Res:	**Post:** s is destroyed and the allocated space is freed

size(s)

Data: s	**Pre:** $s \in S$
Res: n	**Post:** $n \in$ Integer $n = \lvert s \rvert$, the number of keys/key-value pairs in s

empty(s)

Data: s	**Pre:** $s \in S$
Res: b	**Post:** $b \in$ Boolean b = *true*, if $s = \varnothing$ *false*, otherwise

clear(s)

Data: s	**Pre:** $s \in S$
Res: s’	**Post:** $s' = \varnothing$ All the elements are removed.

begin(s)

Data: s	**Pre:** $s \in S$
Res: it	**Post:** it $\in$ Iterator pointing to the first key-value pair Elements are iterable in ascending order by key.

Popoveniuc Mircea
911

containsKey(s, k)

Data: s, k
Res: b

Pre: s ∈ S, k ∈ TKey
Post: b ∈ Boolean
b = *true,* if ∃v ∈ TValue s.t. (k, v) ∈ s,
false, otherwise

getValue(s, k)

Data: s, k
Res: v

Pre: s ∈ S, k ∈ TKey
Post: v ∈ TValue
v = ⊥, if ∄x ∈ TValue s.t. (k, x) ∈ s,
x, otherwise

setValue(s, k, v)

Data: s, k, v
Res: s', it

Pre: s ∈ S, k ∈ TKey, v ∈ TValue
Post: s' = s − (k, x) + (k, v), if ∃x ∈ TValue s.t. (k, x) ∈ s
s + (k, v), otherwise
it ∈ Iterator pointing to (k, v)

at(s, k)

Data: s, k
Res: s', &v (reference)

Pre: s ∈ S, k ∈ TKey
Post: s' = s, if ∃x ∈ TValue s.t. (k, x) ∈ s
s + (k, ⊥), otherwise
v ∈ TValue
v = x, if ∃x ∈ TValue s.t. (k, x) ∈ s,
⊥, otherwise

insert(s, k, v)

Data: s, k, v
Res: s', it

Pre: s ∈ S, k ∈ TKey, v ∈ TValue
Post: s' = s, if ∃x ∈ TValue s.t. (k, x) ∈ s
s + (k, v), otherwise
it ∈ Iterator pointing to (k, x) first case, to (k, v) otherwise

erase(s, k)

Data: s, k
Res: s'

Pre: s ∈ S, k ∈ TKey
Post: s' = s, if ∄x ∈ TValue s.t. (k, x) ∈ s,
s − (k, x), otherwise

erase(s, it)

Data: s, it
Res: s'

Pre: s ∈ S, it ∈ Iterator
Post: s' = s, if it is not a valid iterator
s − (k, v), otherwise, where (k, v) is pointed by it

Sorted Map Iterator

constructor(n)

Data: n
Res: it

Pre: n is "node" from the representation of a sorted map
Post: it ∈ SortedMap<TKey, TValue>::Iterator

Popoveniuc Mircea
911

valid(it)

Data: it
Res: b

Pre: it ∈ SortedMap<TKey, TValue>::Iterator
Post: b ∈ Boolean
b = *true,* if it is valid,
false, otherwise

next(it)

Data: it
Res: it'

Pre: it ∈ SortedMap<TKey, TValue>::Iterator
Post: it' ∈ SortedMap<TKey, TValue>::Iterator
it' = iterator to the next element, which can be NULL if it points to the last element

getElement(it)

Data: it
Res: p

Pre: it ∈ SortedMap<TKey, TValue>::Iterator
Post: p ∈ <TKey, TValue>
p = pair pointed by it

key(it)

Data: it
Res: k

Pre: it ∈ SortedMap<TKey, TValue>::Iterator
Post: k ∈ TKey
k = key from the pair pointed by it

value(it)

Data: it
Res: v

Pre: it ∈ SortedMap<TKey, TValue>::Iterator
Post: v ∈ TValue
v = value from the pair pointed by it

Popoveniuc Mircea
911

Sorted Map Interface

```
template<typename TFirst, typename TSecond>
struct Pair {
	TFirst first;
	TSecond second;

	Pair(TFirst _first, TSecond _second);
	bool operator<(const Pair<TFirst, TSecond>& that) const;
	bool operator>(const Pair<TFirst, TSecond>& that) const;
	bool operator==(const Pair<TFirst, TSecond>& that) const;
};

template <typename TKey, typename TValue>
class SortedMap {
public:
	class Iterator; // see below

	SortedMap();
	SortedMap(const SortedMap<TKey, TValue>& that);
	SortedMap& operator=(const SortedMap<TKey, TValue>& that);
	~SortedMap();

	bool empty() const;
	int size() const;
	void clear();

	Iterator begin();

	bool containsKey(const TKey key);
	TValue getValue(const TKey key);
	Iterator setValue(const TKey key, const TValue value);
	TValue& operator[](const TKey key);
	TValue& at(const TKey key);
	Iterator insert(const TKey key, const TValue value);
	void erase(const TKey key);
	void erase(Iterator& it);
};
```

Popoveniuc Mircea
911

Sorted Map Iterator Interface

```
template <typename TKey, typename TValue>
class SortedMap<TKey, TValue>::Iterator {
public:
	friend SortedMap;

	Iterator(Node* node);
	Iterator(const Iterator& that);
	Iterator& operator=(const Iterator& that);
	~Iterator();

	bool valid() const;
	Iterator next() const;
	Iterator& operator++();
	TKey key();
	TValue& value();
	Pair<TKey, TValue> getElement();
};
```

Popoveniuc Mircea
911

Sorted Map Representation (with Linked List)

```
template<typename TInfo>
struct LinkedList {
	public:
		struct Node {
				TInfo info;
				Node* next;

				Node(TInfo _info, Node* _next = nullptr) :
						info(_info), next(_next) {
				}
		};

		LinkedList();
		~LinkedList();

		Node* append(TInfo info);
		Node* insert(Node* node, TInfo info);
		Node* prepend(TInfo info);

		TInfo remove(Node* node);

		int size() const;

		Node* begin();
		Node* last();

	private:
		Node* begin_;
		Node* last_;
		int size_;
};

template <typename TKey, typename TValue>
class SortedMapWithLinkedList {
private:
	LinkedList<Pair<TKey, TValue>>* elements_;
};
```

Popoveniuc Mircea
911

Sorted Map Operation Design (with Linked List)

```
- constructor()        { O(1) }
- copy_constructor(s) { O(N) }
- destructor(s)        { O(N) }
- size(s)              { O(1) }
- empty(s)             { O(1) }
- clear(s)             { O(N) }
- begin(s)             { O(1) }
- containsKey(s, k)    { O(N) }
```

```
subalgorithm containsKey is
	input: s sorted map, k key
	output: true/false
	if empty(s) = true then
		return false
	end_if
	for each iterator current in s do
		if key(current) > k then
			break for
		else if key(current) = k then
			return true
		end_if
	return false
```

```
- getValue(s, k)       { O(N) }
```

```
subalgorithm getValue is
	input: s sorted map, k key
	output: v corresponding value
	for each iterator current in s do
		if key(current) > k then
			break for
		else if key(current) = k then
			return value(current)
		end_if
	return ⊥
```

```
- setValue(s, k, v)    { O(N) }
- at(s, k)             { O(N) }
- insert(s, k, v)      { O(N) }
```

```
subalgorithm insert is
	input: s sorted map, k key, v value
	output: iterator it
	if empty(s) = true then
		append(s.elements, <k,v>)
		return begin(s)
	end_if
```

```
	if key(begin(s)) > k then
		prepend(s.elements, <k,v>)
		return begin(s)
	end_if
	for each iterator current in s do
		if valid(next(current) = false or key(next(current)) > k then
			break for
		end_if
	end_for
	if key(current) = k then
		return current
	end_if
	insert(s.elements, current.node, <k,v>)
	return next(current)

subalgorithm append is
	input: l linked list, info information for new node
	output: pointer to the newly added node
	; node is added at the end of the list
	new_node <- new node
	new_node->info <- info
	new_node->next <- NULL
	if l.begin = NULL then
		l.begin <- new_node
	else
		l.last->next <- new_node
	end_if
	l.last <- new_node
	l.size <- l.size + 1
	return new_node

subalgorithm prepend is
	input: l linked list, info information for new node
	output: pointer to the newly added node
	; node is added at the beginning of the list
	new_node <- new node
	new_node->info <- info
	new_node->next <- l.begin
	l.begin <- new_node
	l.size <- l.size + 1
	return new_node

subalgorithm insert is
	input: l linked list, n node in linked list, info information for new node
	output: pointer to the newly added node
	; node is added after node n
	new_node <- new node
	new_node->info <- info
	new_node->next <- n->next
	n->next = new_node
	if n is l.last then
```

Popoveniuc Mircea
911

```
        l.last = new_node
    end_if
    l.size <- l.size + 1
    return new_node
```

- **erase(s, k)** { O(N) }

```
subalgorithm erase is
    input: s sorted map, k key
    output: pair with key k is erased from s, if it exists
    if containsKey(s, k) = false then
        return
    for each iterator current in s do
        if key(current) = k then
            remove(s.elements, current.node)
            return
        end_if
    end_for

subalgorithm remove is
    input: l linked list, n node in l
    output: n is removed from l
    if n is not l.begin then
        previous <- l.begin
        while previous->next is not n do
            previous <- previous->next
        end_while
        previous->next <- n->next
    else
        l.begin <- n->next
    end_if
    delete n
```

- **erase(s, it)** { O(N) }

```
subalgorithm erase is
    input: s sorted map, it iterator
    output: pair pointed by it is erased from s, if it is valid
    if valid(it) = false then
        return
    remove(s.elements, it.node)
```

Popoveniuc Mircea
911

Sorted Map Representation (with Balanced Tree / Treap)

```
template<typename TInfo>
struct Treap {
	public:
		struct Node {
			TInfo info;
			int priority;
			Node* father;
			Node* left_son;
			Node* right_son;

			Node(TInfo _info, int _priority) :
				info(_info), priority(_priority),
father(nullptr), left_son(nullptr), right_son(nullptr) {
			}
		};

		Treap();
		~Treap();

		int size() const;

		Node* find(TInfo info);
		Node* insert(TInfo info);
		void erase(Node* node);
		Node* root();

	private:
		Node* root_;
		int size_;
		std::random_device rand_;

		Node* find(Node* root, TInfo info);
		void insert(Node*& root, Node*& new_node);
		void erase(Node*& root, Node*& new_node);
		void balance(Node*& root);
		void rotate_left(Node*& root);
		void rotate_right(Node*& root);
};

template <typename TKey, typename TValue>
class SortedMapWithBalancedTree {
private:
	Treap<Pair<TKey, TValue>>* elements_;
};
```

Popoveniuc Mircea
911

Sorted Map Operation Design (with Balanced Tree / Treap)

- **constructor()** { O(1) }
- **copy_constructor(s)** { O(N log N) }
- **destructor(s)** { O(N log N) }
- **size(s)** { O(1) }
- **empty(s)** { O(1) }
- **clear(s)** { O(N log N) }
- **begin(s)** { O(log N) }

```
subalgorithm begin is
      input: s sorted map
      output: iterator pointing to first node of tree in inorder search
      current <- elements.root
      while current->left is not NULL then
            current <- current->left
      return Iterator(current)
```

- **containsKey(s, k)** { O(log N) }

```
subalgorithm containsKey is
      input: s sorted map, k key
      output: true/false
      ; note: equality between information (as pairs) from two nodes is equality of
keys, regardless of the value
      if find(s.elements, <k, ⊥>) is NULL then
            return false
      else
            return true
      end_if

subalgorithm find is
      input: t treap, info information to be found
      output: pointer to the corresponding node
      return find(t.begin, info)

subalgorithm find is
      input: root treap-node, info information to be found
      output: pointer to the corresponding node
      if root is NULL then
            return root
      end_if
      if root->info > info then
            return find(root->left, info)
      else if root->info < info then
            return find(root->right, info)
      end_if
      return root
```

Popoveniuc Mircea
911

- getValue(s, k)   { O(log N) }
- setValue(s, k, v)   { O(log N) }
- at(s, k)   { O(log N) }
- insert(s, k, v)   { O(log N) }

```
subalgorithm insert is
	input: s sorted map, k key, v value
	output: iterator it
	if containsKey(s, k) = true then
		return Iterator(find(s.elements, <k, ⊥>))
	end_if
	return Iterator(insert(s.element, , <k, v>))

subalgorithm insert is
	input: t treap, info information to be added
	output: pointer to the corresponding node
	new_node <- new node
	new_node->info = info
	new_node->priority = random()
	if t.root = nullptr then
		t.root <- new_node
		t.size <- t.size + 1
		return new_node
	end_if
	t.size <- t.size + 1
	return insert(t.begin, info)

subalgorithm insert is
	input: root reference to treap-node, node treap-node to be added
	output: node is added and subtreap is balanced
	if root->info > node->info then
		insert(root->left, node)
	else
		root->left = node
		node->father = root
	end_if
	if root->info < node->info then
		insert(root->right, node)
	else
		root->right = node
		node->father = root
	end_if
	balance(root)

subalgorithm balance is
	input: root reference to treap-node
	output: root is balanced and might change
	if root->left is not NULL and root->left->priority < root->priority then
		rotate_right(root)
	end_if
	if root->right is not NULL and root->right->priority < root->priority then
```

```
        rotate_left(root)
    end_if

subalgorithm rotate_left is
    input: root reference to treap-node
    output: subtreap is rotated to left and root becomes its right son
    right_son <- root->right
    right_son->father <- root->father
    root->father <- right_son
    root->right <- right_son->left
    right_son->left <- root
    root <- right_son

subalgorithm rotate_right is similar to rotate_left
```

- **erase(s, k) { O(log N) }**
- **erase(s, it) { O(log N) }**

```
subalgorithm erase is
    input: s sorted map, it iterator
    output: pair pointed by it is erased from s, if it is valid
    if valid(it) = false then
        return
    erase(s.elements, it.node)

subalgorithm erase is
    input: t treap, n treap-node to be erased
    output: n is erased from t
    if n is NULL then
        return
    end_if
    erase(t.root, n)
    delete n
    t.size <- t.size - 1

subalgorithm erase is
    input: root reference to treap-node, n treap-node to be erased
    output: n is erased from subtreap and subtreap is balanced
    if root is NULL then
        return
    end_if
    if root->info > n->info then
        erase(root->left, n)
    else if root->info < n->info then
        erase(root->right, n)
    else
        if root->left is NULL and root->right is NULL then
            root <- NULL
            return
        end_if
```

```
            if root->left is NULL or (root->right is not NULL and
root->right->priority < root->left->priority) then
                  rotate_left(root)
                  erase(root, n)
                  return
            end_if
            if root->right is NULL or (root->left is not NULL and
root->left->priority < root->right->priority) then
                  rotate_right(root)
                  erase(root, n)
                  return
            end_if
      end_if
```

Popoveniuc Mircea
911

Solved Problem

Consider the next general problem:
Find all words in a dictionary that can be formed with the letters of a given word.
E.g.: orchestra / carthorse
steak / skate

Find all words in a dictionary that can be formed with the letters of a given word: Use the same letters, all the letters, but letters can have different frequencies.

In the solution, it was considered that the dictionary contains only lowercase letters of the English alphabet.

Solution:

Read all words and encode every word in the following way:
For a certain word, use a 32-bit integer where each bit marks the presence of a certain letter in the word: the 0th bit is 1 if 'a' is present or 0 otherwise, the 1st bit is 1 if 'b' is present or 0 otherwise, ..., the 25th bit is 1 if 'z' is present or 0 otherwise.

E.g.: word "roof": 'f' is present, corresponding to the 5th bit
'o' is present, corresponding to the 14th bit
'r' is present, corresponding to the 17th bit
So the corresponding code for "roof" is $2^5 + 2^{14} + 2^{17} = 147488$
Note that the word "for" has the same encoding and respects the criteria that it uses the same letters and all of them (despite the different frequencies).

The encoding is based on the fact that only lowercase letters are present in the words, so only 26 bits are required for the code.

We use a sorted map to associate a code to a list of words.

```
int -> std::vector<string>
```

When a word is given, use the corresponding code to retrieve the list of words satisfying the required criteria and print it.

Input/Output specification and test sets

When running the executable, the program will ask a filename with words and a word.
After building the dictionary from the given file and finding the good words, the program will print the god words in the console and the execution will stop.

The problem was solved using `SortedMapWithLinkedList`, `SortedMapWithBalancedTree` (with a treap) and `STLMapWrapper` that wraps `std::map` from STL in order to use the interface in solving the problem (only the operations that are used were wrapped).

Popoveniuc Mircea
911

The following test data is provided:

Filename	**Description**
words1.txt	File containing 10 words.
words2.txt	File containing 1000 words.
words3.txt	File containing 6583 words.
words4.txt	File containing 10000 words.
words5.txt	File containing 350630 words.

The following execution times (in seconds) have been obtained on average for building the dictionary based on the read words and for retrieving and printing the good words:

Filename	**With linked list**	**With balanced tree**	**With std::map**
words1.txt	Building: 0.00002	Building: 0.00061	Building: 0.00003
	Printing: 0.00000	Printing: 0.00001	Printing: 0.00000
words2.txt	Building: 0.03156	Building: 0.01787	Building: 0.00358
	Printing: 0.00001	Printing: 0.00001	Printing: 0.00000
words3.txt	Building: 2.06023	Building: 0.14216	Building: 0.02561
	Printing: 0.00057	Printing: 0.00003	Printing: 0.00001
words4.txt	Building: 7.52987	Building: 0.23668	Building: 0.04493
	Printing: 0.00060	Printing: 0.00003	Printing: 0.00001
words5.txt	Building: over 60	Building: 8.95641	Building: 1.45684
	Printing: over 60	Printing: 0.00004	Printing: 0.00002

As expected, the implementation with linked list performed poorly on bigger data compared to the implementation with balanced tree due to the increased complexity of the operations. The implementation with balanced tree, despite having the same theoretical asymptotical complexity for most of its operations with the std::map, is outperformed by std::map due to the fact that the red-black tree (used in std::map's implementation) is better balanced than the treap (which is almost balanced and has a higher height, although the height is proportional to the logarithm of the size of the tree).

The conclusion is that balanced tree is a more efficient representation for the sorted map.