Legenda

ADT: Set
1. **Class particularities**
2. **Specifications**
3. **Interface**
4. **Representations**
5. **Diagram**

Dynamic vector
6. **Interface**
7. **Complexity**
8. **Algorithm design**

Hash table
9. **Interface**
10. **Complexity**
11. **Algorithm design**

STL Set
12. **Interface**
13. **Complexity**
14. **Algorithm design**

Problem
15. **Statement**
16. **Input/output**
17. **Test data**
18. **The idea**
19. **Execution time**

About
20. **Best DS for this ADT**
21. **Best ADT for this problem**
22. **Importance**

1. **Set particularities**
   - domain: set of all elements of type TElement
   - description: unique, comparable elements in undefined order
   - data: TElement – type of elements

2. **Set specifications (Set functions)**
   - Constructor(S)
     - description: creates a new instant of Set<TElement>
     - data: S
     - result: -
     - pre: S belongs to Set<TElement>
     - post: new set S created
   - Destructor(S)
     - description: deallocates the used memory
     - data: S
     - result: -
     - pre: S belongs to Set<TElement>
     - post: -
   - isEmpty(S)
     - description: checks whether S is empty or not
     - data: S
     - result: true / false
     - pre: S belongs to Set<TElement>
     - post: true – if S is empty, false - otherwise
   - size(S)
     - description: finds the number of elements of S
     - data: S
     - result: number of elements
     - pre: S belongs to Set<TElement>
     - post: number of elements of S
   - contains(S, e)
     - description: determines whether S contains element e or not
     - data: S, e
     - result: true / false
     - pre: S belongs to Set<TElement>, e is of type TElement
     - post: true – if S contains e, false - otherwise
   - insert(S, e)
     - description: adds element e to S
     - data: S, e
     - result: true / false
     - pre: S belongs to Set<TElement>, e is of type TElement
     - post: true – if e was added to S, false - otherwise
   - erase(S, it)
     - description: removes element at position it in S (by iterator)
     - data: S, it
     - result: true / false
     - pre: S belongs to Set<TElement>, it valid iterator // check isValid(it) from Iterator
     - post: true – if element at position it was removed from S, false - otherwise

- remove(S, e)
  - description: removes element e from S
  - data: S, e
  - result: true / false
  - pre: S belongs to Set<TElement>, e is of type TElement
  - post: true – if e was removed from S, false – otherwise

Set specifications (Iterator functions)

- Iterator(S)
  - description: returns an enumerator which iterates through set S
  - data: S
  - result: it
  - pre: S belongs to Set<TElement>
  - post: it - enumerator
- get(S, it)
  - description: returns the element at position it in S
  - data: S, it
  - result: e
  - pre: S belongs to Set<TElement>, it – valid iterator
  - post: e – element on position it
- next(S, it)
  - description: moves to the next element in S
  - data: S, it
  - result: -
  - pre: S belongs to Set<TElement>, it – valid iterator
  - post: it += 1
- isValid(S, it)
  - description: checks whether the position it is valid or not
  - data: S, it
  - result: true / false
  - pre: S belongs to Set<TElement>, it belongs to Z
  - post: true – it it is valid, false - otherwise

3. **Set interface (Set)**

```
Set Constructor();
void Destructor();
bool isEmpty();
int size();
bool contains(TElement e);
bool insert(TElement e);
bool erase(int it);
bool remove(TElement e);
```

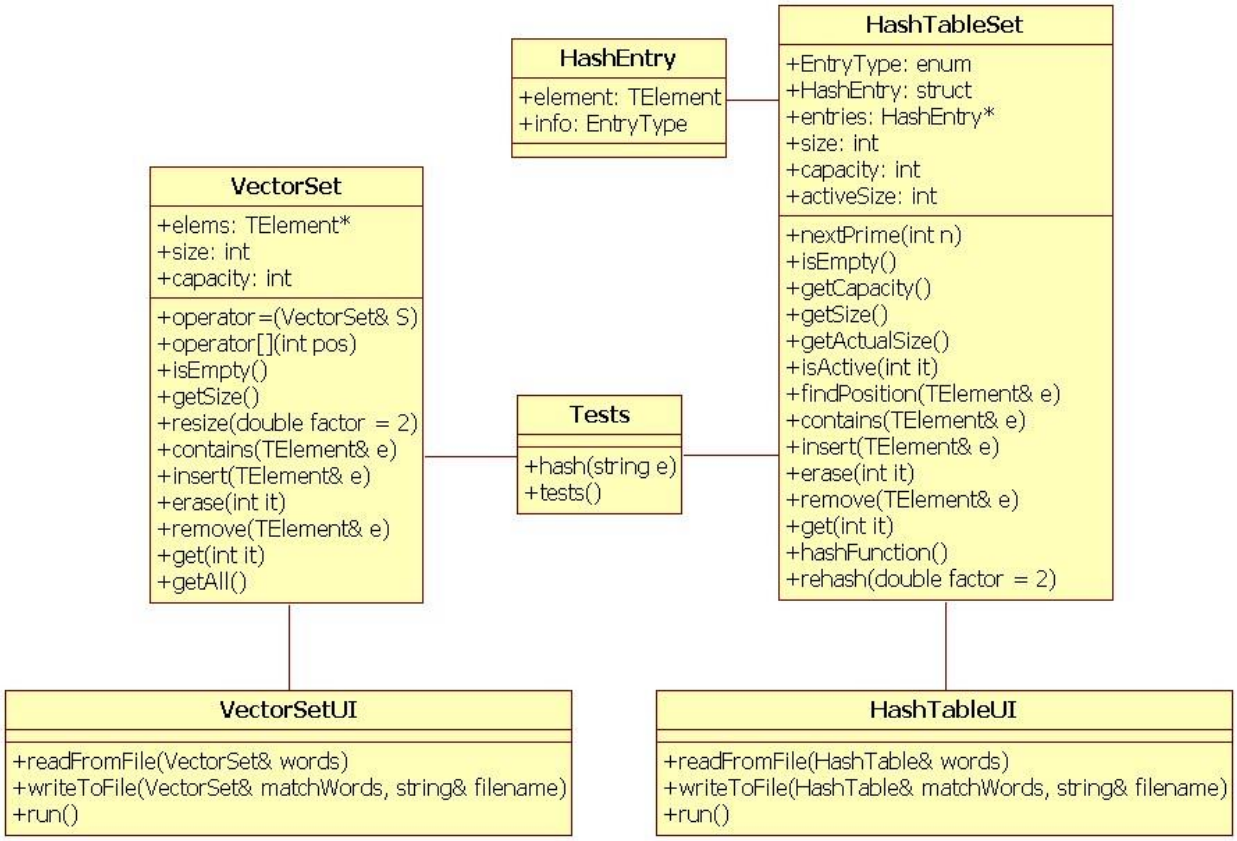
Set interface (Iterator)

```
TElement get(int it);
void next(int it);
```

4. Set representations

- Dynamic vector
  - ➔ a dynamically allocated array, its size and its capacity
- Hash table with open addressing
  - ➔ a struct of a couple information and element, a dynamically allocated array of structs
  - ➔ its size, its actual size (active – not lazy deleted items) and its capacity

5.



6. Dynamic vector interface

```
VectorSet(int capacity = 10); // constructor
VectorSet(VectorSet& S); // copy constructor
~VectorSet(); // destructor
VectorSet& operator=(VectorSet& S); // assignment operator
TElement& operator[](int pos); // position operator
bool isEmpty(); // is empty validator
int getSize(); // size getter
bool contains(TElement& e); // contains function
bool insert(TElement& e); // adder
bool erase(int it); // remover (by iterator)
bool remove(TElement& e); // remover (by value)
TElement& get(int it); // element getter (by iterator)
TElement* getAll(); // elements getter
```

Iterator interface

```
IteratorVectorSet(VectorSet<TElement>& s) : mySet{ s } {}; // constructor
TElement* begin(); // iterator pointing to the first element
TElement* end(); // iterator pointing AFTER the last element
bool isValid(int it); // iterator validator
void next(int it); // iterator increaser
```

7. **Dynamic vector complexity** (let n = vector size)
   - checking for element: O(n)
   - inserting: O(n)
   - erasing: O(n)
   - removing: O(n)
   - resizing: O(n)

Justification:
- checking means checking all positions [O(n)];
- inserting implies checking [O(n)];
- erasing implies moving the elements on the right of the one to be erased one position to the left [O(n)];
- removing means checking and erasing [O(n)];
- resizing implies coping all elements [O(n)];

8. **Algorithm design - pseudocode**
   - contains(TElement& e)

```
it = 0;
while (it != size)
        if (e == elems[it])
                return true
        else
                it++
        endif
endwhile
return false
```

   - insert(TElement& e)

```
if (size == capacity)
        resize()
endif
if (contains(e))
        return false
endif
elems[size] = e
size++
return true
```

9. **Hash table interface**

```
int nextPrime(int n);
HashTable(std::function<int(TElement)> hashFct, int capacity = 101); // constructor
~HashTable(); // destructor
bool isEmpty(); // is empty validator
```

```
int getCapacity(); // capacity getter
int getSize(); // size getter
int getActualSize(); // size getter (with deleted)
bool isActive(int it); // element verifier
bool contains(TElement& e); // contains function
bool insert(TElement& e); // adder
bool erase(int it); // remover (by iterator)
bool remove(TElement& e); // remover (by value)
TElement& get(int it); // element getter (by iterator)
```

Iterator interface

```
IteratorHashTable(HashTable<TElement>& s) : mySet{ s } {}; // constructor
TElement* begin(); // iterator begin()
TElement* end(); // iterator end()
bool isValid(int it); // iterator validator
void next(int it); // iterator increaser
```

10. Hash table complexity (let n = hash table capacity ~= 2*hash table size)

- checking for element: undetermined, usually O(1)
- inserting: undetermined, usually O(1)
- erasing: O(1)
- removing: undetermined, usually O(1)
- resizing: O(n)

Justification:

- checking means finding the right position of an element after computing the result of the hashing function mod n; the time is undetermined, somewhere between O(1) and O(n); as this implementation comes with QUADRATIC PROBING and substitutes its hashing function with the one given by the user, the one most appropriate for the type of the element, the complexity is usually O(1);
- inserting implies checking [undetermined, usually O(1)];
- erasing implies lazy deleting the element at a given position (marking it DELETED) [O(1)];
- removing means checking and erasing [undetermined, usually O(1)];
- resizing implies coping all elements [O(n)];

11. Algorithm design – C++ with explications

```
HashTable(std::function<int(TElement)> hashFct, int capacity = 101) : hashFunction{ hashFct } {
    // a hashing function will be passed by to hashFunction when initializing a Set
    // this allows using any hashing function; maybe an appropriate one to the type of the elements
    this->capacity = nextPrime(capacity);
    this->size = 0; this->activeSize = 0;
    this->entries = new HashEntry[this->capacity];
    // memory is allocated for the array of elements
    for (int i = 0; i < this->capacity; i++) {
        this->entries[i] = HashEntry{};
        // every entry is initialized empty
    }
}
```

```
rehash(double factor = 2) {
	int oldCapacity = this->capacity;
	this->capacity = nextPrime(oldCapacity*2);
	// the capacity must always be a prime (Quadratic probing)
	HashEntry* oldEntries = entries;
	// the array of entries is copied
	this->entries = new HashEntry[this->capacity];
	// new memory is reallocated
	for (int i = 0; i < this->capacity; i++) {
		this->entries[i] = HashEntry{};
		// every entry is initialized empty
	}
	this->size = 0; this->activeSize = 0;
	for (int i = 0; i < oldCapacity; i++) {
		this->insert(oldEntries[i].element);
		// old entries are copied to the new array
	}
	delete[] oldEntries;
	// auxiliary array for coping is deallocated
}

findPosition(TElement& e) {
	int offset = 1, pos = std::abs(this->hashFunction(e)) % this->capacity;
	// the position is the return of the hashing function % the hash table's capacity
	while (this->entries[pos].info != EMPTY && this->entries[pos].element != e) {
		pos += offset*offset;
		offset++;
		// if the position is already occupied go to the next one
		if (pos >= this->capacity)
			pos -= this->capacity;
			// if we reached the end, go back and search from the beginning
	}
	return pos;
}

insert(TElement& e) {
	int pos = this->findPosition(e);
	// find the position
	if (this->isActive(pos))
		return false;
		// if it is active it means the element already exists (we skip adding it)
	this->entries[pos] = HashEntry(e, ACTIVE);
	this->size++; this->activeSize++;
	// the entry is marked active and size increases
	if (this->size >= this->capacity / 2)
		this->rehash();
		// rehash if needed
	return true;
}
```

12. STL Set interface

- this was not at all implemented by me, so I just give a reference: http://www.cplusplus.com/reference/set/set/

13. STL Set complexity (let n = size of the red-black tree)

- checking for element: O(log(n))
- inserting: O(log(n))
- erasing: O(log(n))
- removing: O(log(n))
- restructuring(recoloring): O(1)

Justification:

- checking means finding the element down the branches of a red-black tree; this has the worst case O(log(n)) but it is usually even less than that;
- inserting implies checking and restructuring [O(log(n))];
- erasing implies restructuring [O(log(n))];
- removing means checking and erasing [O(log(n))];
- restructuring implies resolving the perturbations of two consecutive red edges [O(1)];

14. Algorithm design

- this was not at all implemented by me, so I just give a hint:
  red – **black** trees

15. Problem statement

Find all words in a dictionary that can be formed with letters of a given word. Use the same letters, but not necessarily all of them. Letters can have different frequencies.

16. Input/output

- two input files: one of 7 words, one of 10000 words; words can have any length (well, excluding 0 for the logic reasons regarding the task), but they are considered to be exactly one word on each line in the files to be read; the active current file is the big one;
- a word read from keyboard (the given word)
- one output file, chose by the user, in which the solution of the problem will be written (the words which were found)

Running example

input file: abcde/edabc/abcdeacbaec/ababecaabbcbaec/abcdef/abcfdeacfbaec/abcfgdeacfbaec/
given word: abdec
written file: abcde/edabc/abcdeacbaec/ababecaabbcbaec/s

17. Test data

- the small file, the one of 7 words, named "dictionary", is created for testing quickly if the idea is appropriate and the algorithm is good; it can be modified, and it is designed to be modified – words can be changed, added, deleted to check the program on different small data, of which the user knows the solution
- the big file, the one of 10000 different words, named "bigDictionary", cannot is there to check how well and how fast the program runs; this is the one that digs deeper and determines whether the implemented ADT is good and appropriate and not only the problem's solution

18. The idea

a. process the file: create a set of strings to store the words; insert each word in it
b. process the given word: read it; create another set, this one of chars, in which store all the letters of the given word
c. search for matches: create yet another set, of strings, in which insert all the matching words, in other words, the ones formed only by letters from the set of chars
d. write to file: write every word from the set of matches in the file
e. that's it

19. Execution time

- *input file:* dictionary.txt *given word:* abcde

	Reading from file	Reading word	Matching	Writing to file
Dynamic vector	0.00s	0.00s	0.00s	0.01s
Hash table	0.00s	0.00s	0.00s	0.00s
STL Set	0.00s	0.00s	0.00s	0.01s

- *input file:* bigDictionary.txt *given word:* bacteria

	Reading from file	Reading word	Matching	Writing to file
Dynamic vector	121.44s	0.00s	0.22s	0.20s
Hash table	5.15s	0.00s	0.40s	0.20s
STL Set	1.66s	0.00s	1.36s	0.10s

- *input file:* bigDictionary.txt *given word:* iamthelongestwordinthedictionaryyuhooooooooo

	Reading from file	Reading word	Matching	Writing to file
Dynamic vector	119.38s	0.00s	14.01s	0.07s
Hash table	5.19s	0.00s	2.72s	0.10s
STL Set	1.61s	0.00s	1.91s	0.17s

20. Best DS for set: red-black tree

- consider both ordered set and unordered set
- argument: however we choose a DS which is not a tree, either the searching for an element xor the inserting can actually cost O(n), which is a lot;
  the hashtable was a very good way to tackle this problem, because few elements did cost that much and much more costed O(1) – however, the downsize, the resizing will always cost O(n);
  so, how to minimize this one cost which is always big? use a tree, a balancing it constantly with respect to some coloring rules; this way, all operations are bigger a bit than O(1) but a lot smaller than O(n); they are O(log(n)); only the restructuring itself takes O(1);
- contra-argument: always, the input data decides, so depending on what we use the set for, there are different methods to implement it, that's the reason to learn and understand them

Best DS for unordered set: hash table

➢ red-black tree not an option

- open addressing: doesn't require much memory; erasing costs surely O(1) and all the other operations, except resizing, in many cases will cost O(1) also; resizing takes time, but, if the size of the input data is known, its factor can be chosen appropriate
- chaining: it's commonly a bit faster; if the size of the input data is not known, it's definitely what to choose – no resizing needed, adding more data won't make it much slower

21. Best ADT for this problem: trie

- here I have been discussing about sets, but, ultimately, I think that neither a red-black tree or a hash table will do the job in the best possible way
- the best solution I can think of is a trie, a specially designed tree to serve the purpose of a dictionary

22. Importance: why is DSA important? what did I learn?

- worst case scenario matters, but the majority of cases, so the particular input data matters more
- every way of implementing something has advantages and disadvantages, understanding them and knowing them helps coming with the best solution for the arising problem
- when data is big, difference is big, and nowadays data is big
- coding is interesting when understanding the depths and the evolution of thinking in programming, not when copy-pasting