****

*Data Structures and Algorithms*

*Balloons and Arrows*

**Table of contents**

**Page 3. Problem related topics**

**Page 4. Abstract Data Type Definition**

**Page 5. Data Structure Definitions**

**Page 7. Problem related Class and Function Definitions**

**Page 8. Input / Output Specificiations**

**Page 9. File Hierarchy**

**Page 10. Solving Algorithm**

**Page 12. Pseudocode**

**Page 14. Pros and Cons for the DS implemented**

**Page 15. Run examples**

**Page 17. Output examples**

**Page 18. Misc Questions**

**Problem related topics**

**Problem definition**:

Given **n** balloons, determine the **smallest** number of arrows to **break all** the balloons. It is considered that the arrows are launched **vertically**. (**Solution should also present the arrow positions**)

**Theme:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nr.** | **ADT** | **Representation 1** | **Representation 2** |
| 3. | Sorted List, Sorted List Iterator, FWD | Dynamic Vector | Singly Linked List |

**General rules used in implementation**:

* All parameter variables **start** with capital letter and are **CamelCase**. (e.g. SList, Node, Other)
* **All private** or **protected** variables have an **underscore** suffix. (e.g. count\_, current\_)
* **Some** Operators, Constructors, Destructors, Setters or Getters **are not specified** as the code itself is **easy to understand.**
* **Some** class definitions also contain **other** class definitions inside them. For those, the functions are defined in the **same** file. (see comments for more details)
* For **more** details read each file’s **Top Comment**.

**Compilation guidelines**:

Project was created using Visual Studio 2013. In order to view the project, ‘**balloons.sln**’ file must be opened, which can be found in the **root folder** of the project. The project is configured to run on almost any machine that has MSVisualStudio 2013 installed, and the latest VC++ Redistributables.

**Abstract Data Type Definition**

<typename T> class **AbstractSortedList**:

* Protected attributes:
  + count\_ (int)
* Public functions:
  1. Virtual T getAtIndex(const int& Index);
  2. Virtual void add(const T& Object);
  3. Virtual void removeAtIndex(const int& Index);
  4. Virtual bool contains(const T& Key);
  5. Virtual void clear();
  6. Virtual void size();

1. Get element from index given as parameter
2. Add element into the list (sorted ascending)
3. Remove element from index given as parameter
4. Check if Key is contained in the list (true or false)
5. Clear the list (count resets and list is freed, if allocated)
6. Return this->count\_

<typename T, typename O> class **AbstractSortedListIterator**:

* Protected attributes
  + T\* current\_
* Public functions:
  1. Virtual bool isValid();
  2. Virtual bool hasNext();
  3. Virtual void operator++();
  4. Virtual O& operator\*() const;

1. Check if iterator is valid
2. Check if iterator has a next element
3. Go to the next element in the container
4. Return the data in the container

**Data Structures Definition**

<typename Object> class **SortedSLList**:

* Class inherts from **AbstractSortedList<Object>** and overrides all functions
* Inherited functions:
  + getAtIndex()
  + add()
  + removeAtIndex()
  + contains()
  + clear()
  + size()
* Extra functions:
  + getFront()
  + getBack()
  + removeFront()
  + removeBack()
  + begin()
  + end()
* <typename Object> class **SLNode**:
  + Protected attributes:
    - data\_ (Object)
    - next\_ (SLNode \*)
  + Public functions:
    - SLNode& operator=(const SLNode& Node);
    - Object& getData();
* class **Iterator**:
  + Class inherits from **AbstractSortedListIterator**<SLNode, Object>
  + Overloads operators: ++, ++(int), ==, !=, =, \*.
* UnitTest – See “~\unittest\SortedSLListTest.cpp”

**Data Structures Definition**

<typename Object> class **SortedDynamicVector**:

* Class inherts from **AbstractSortedList<Object>** and overrides all functions
* Inherited functions:
  + getAtIndex()
  + add()
  + removeAtIndex()
  + contains()
  + clear()
  + size()
* Extra functions:
  + getFront()
  + getBack()
  + removeFront()
  + removeBack()
  + begin()
  + end()
* class **Iterator**:
  + Class inherits from **AbstractSortedListIterator**<SLNode, Object>
  + Overloads operators: ++, ++(int), ==, !=, =, \*.
* UnitTest – See “~\unittest\SortedDynamicVectorTest.cpp”

**Problem related Class and Function Definitions**

class **Balloon**:

* Private attributes:
  + xCoordStart\_ (float)
  + xCoordEnd\_ (float)
  + has Arrow\_ (bool)
* Public functions:
  + Overloads operators =, ==, <, >, <=, >=, << (for printing)
  + setXCoordStart()
  + setXCoordEnd()
  + setArrowStatus()
  + getXCoordStart()
  + getXCoordEnd()
  + getArrowStatus()

**Reading data from file**:

* template <typename **Container**>

**int** ReadData(**Container** \*SList, **std::string** InputFilePath);

**Solving balloon and arrow problem**:

* template <typename **Container**, typename **ArrowContainer**>

**int** SolveBalloonsProblem(**Container** \*SList,

**ArrowContainer** \*SList,

**std::string** InputFilePath);

class **BlackBoxTest**:

* Public functions:
  + testSet0()
  + testSet1()
  + testSet2()
* void StartBlackBoxTest() – Defined outside class (will be called in main)

**Input / Output Specifications**

**Input files**:

* **Input files** found in “~\balloons\data\input”
* **Test input files** found in “~\balloons\data\test\input”
* **Row format:** xCoordStart xCoordEnd
* **Example**: (found in data1.in)
  + 1.0 2.3

1.3 1.5

1.6 2.2

1.7 2.2

2.7 4.5

**Output files**:

* **Output files** found in “~\balloons\data\output”
* **Test output files** found in “~\balloons\data\test\output”
* **Output format**:
  + Arrow found at coordinate X
  + Total number of arrows: X
* **Example**: (found in data1.out)
  + ----------------------------------------------------------------

File: .\data\output\data1.out.in

----------------------------------------------------------------

Arrow found at coordinate 1.5

Arrow found at coordinate 2.2

Arrow found at coordinate 4.5

Total number of arrows: 3

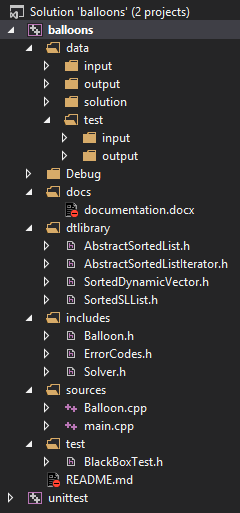
**Solution files**:

* **Solution files** found in “~\balloons\data\solution”
* **Solution format**: (each set of balloons shot by an arrow is delimitated by “-------“
  + -------

1.0 2.3

-------

**File Hierarchy**



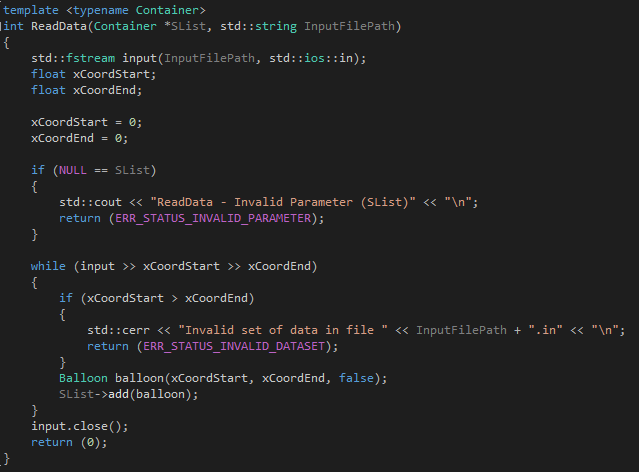
**File hierarchy is as follows**:

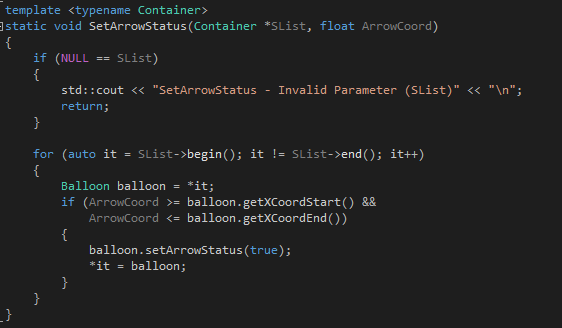
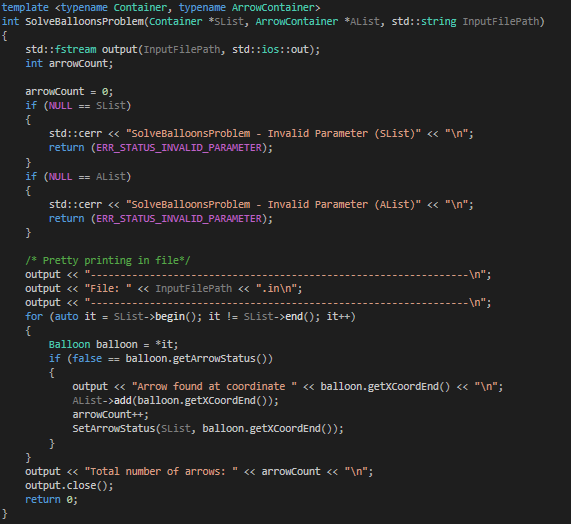
* **data** – Input and output files
  + input – input files
  + output – output files
  + solution – solution files
  + test
    - input – tests
    - output – tests
* **docs** – Documentation file
* **dtlibrary** – Data Type Library
* **includes** – Include files
* **sources** – Source files
* **test** – Test files

**Solving Algorithm**

**Main solving idea**:

* The software reads all the file input, and inserts all elements in the containe, **sorted by the most right coordinate of the balloon** (xCoordEnd). Once the container contains all the balloons, then the solver function takes the first balloon, fires an arrow through the **most right** coordinate of it and calls a function that checks which balloons were shot and toggles their **arrow status** to **True**. Then the function looks for the next balloon which is not shot, and repeats the steps.
* **Steps**:
  + Find first balloon which is not shot
  + Shoot an arrow through xCoordEnd
  + Check which balloons were shot by this arrow
  + Toggle their arrow status to true
  + Repeat –

****

****

**Pseudocode**

**InsertSorted (SortedSLList - add)**:

* if list is empty
  + create new node
  + head becomes the new node
  + tail becomes the new node
* else
  + searchNode <- front
  + prevSearchNode <- front
  + if (Data < searchNode->data)
    - Add in front
  + else
    - while (searchNode is not NULL and Data > searchNode->data)
      * prevSearchNode <- searchNode
      * searchNode <- searchNode->next
    - add at prevSearchNode->next
    - if we are at the end of the list
      * back <- prevSearchNode->next

**InsertSorted (SortedDynamicVector - add)**:

* make sure there is enough space to add element
* while (index < count and data > array[index])
  + index++
* move all elements with one position to the right starting with index to make place for the new element
* array[index] <- Data //(add the new element)
* count++

**Pseudocode**

**Expand vector(SortedDynamicVector - ensureCapacity)**:

* if (capacity >= new capacity)
  + do nothing and return from function
* aux <- array
* capacity <- capacity \* 2
* array <- (array of objects[capacity])
* copy elements from aux to array
* delete/free aux

**Remove At Index(SortedSLList - removeAtIndex)**:

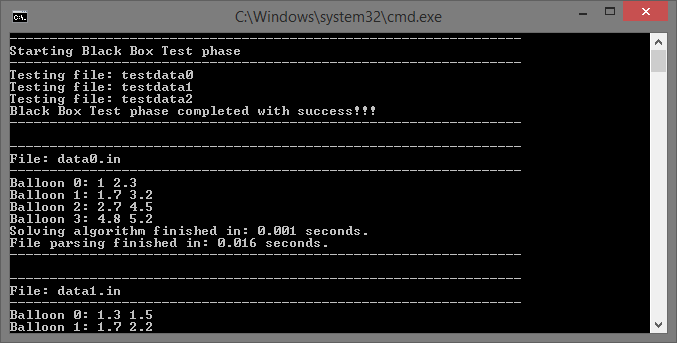
* if (index > count – 1) //we try to remove an object outside the list
  + do nothing
* node <- front
* prevNode <- front
* index <- 0
* while (index < IndexToRemoveAt)
  + prevNode <- node
  + node <- node->next
  + index++
* prevNode->next <- node->next
* delete node
* count--

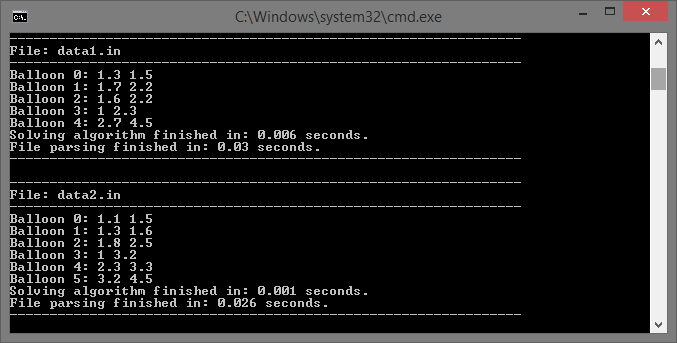
**Pros and Cons for the DS implemented**

* **SortedSingleLinkedList**:
  + Pros:
    - No need for capacity increase (no capacity, can hold unlimited elements)
    - Spread all over heap (there’s no need for a ‘big’ enough block)
  + Cons:
    - Slow index-based access of data (not position based) list must be parsed every time
    - More memory needed for each element (Data + ‘next’ pointer)
* **SortedDynamicVector**:
  + Pros:
    - Less memory needed for each element (Does not hold ‘next’ pointer)
    - Faster index-based access of data (vector must not be parsed, instead we use pointer arithmetic)
  + Cons:
    - Every time size becomes bigger than capacity, the vector must be expanded (big slowdown for large amount of data, vector must be copied every time)
    - When using large amount of data, a large enough memory block must be free for allocation
* **Which ADT is best used for this problem?**
  + Any **SORTED ADT** will perform excellent.
  + For more **time and memory** **efficiency** I would use index-based DS implementation (Dynamic Vector)
  + For **large collection of data** I would use Single Linked List

**Run examples**

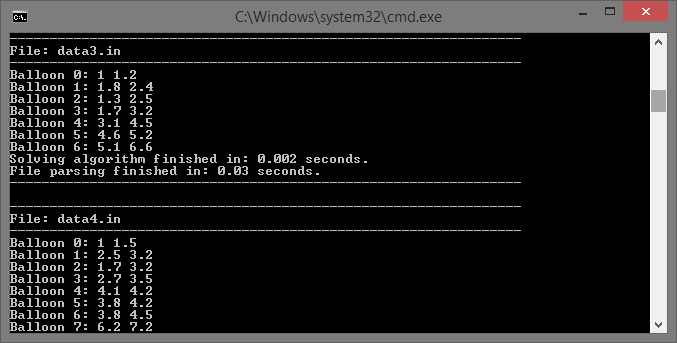
**Run example for the software**:

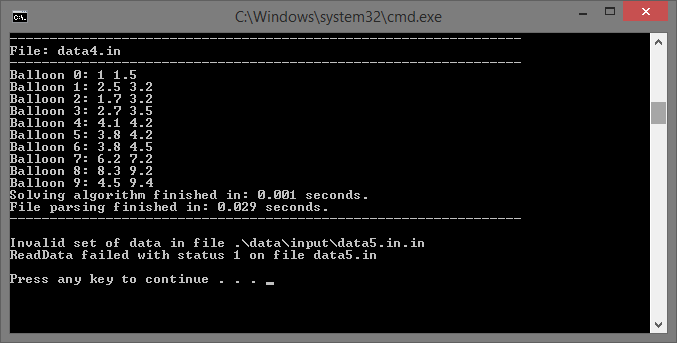




**Run examples**

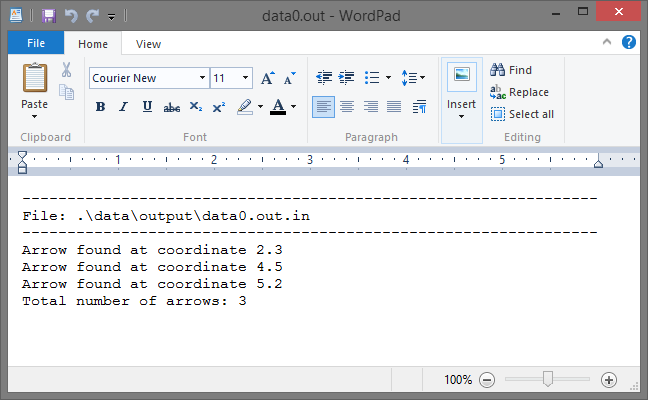
**Run example for the software**:

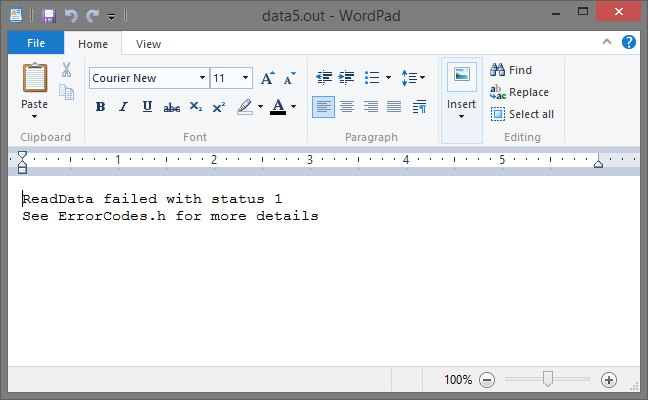




**Output examples**

**Output example for the software (dataX.out)**



****

**Misc Questions**

**What have I learned from this project**?

* I have improved my C++ comprehension skills, learned to use more features of the C++ language and improved my programming skills overall.
* I have improved my Single Linked List and Dynamic Vector knowledge overall.
* I have understood the Greedy algorithm principle.
* I have improved my Visual Studio 2013 IDE knowledge.
* I have understood the UnitTest mechanism offered by Visual Studio 2013.

**What I think about this project**?

* It was a fun project that required a lot of work, which helped me improve my programming skills.

**Why we study Data Structures**?

* In order to properly use memory for problem solving and to avoid unwanted behavior, we must study different Data Structures which can be applied in different contexts. For example, I could use any Sorted Data Structure to solve the ‘Balloons and Arrows’ problem. However, a binary tree would be more efficient for solving different mathematical expressions.

**Why are ADT’s important**?

* ADT’s are a theoretical concept in Computer Science, used in the design and analysis of Algorithms, Data Structures and Software Systems, and do not correspond to specific features of computer languages. They are used to simplify the description of abstract Algorithms, to classify and to evaluate Data Structures.