****

**BLG 335E**

**ANALYSIS OF ALGORITHMS I**

CRN: 10825

**REPORT OF HOMEWORK #2**

Submission Date: 26.11.2013

**STUDENT NAME: TUĞRUL YATAĞAN**

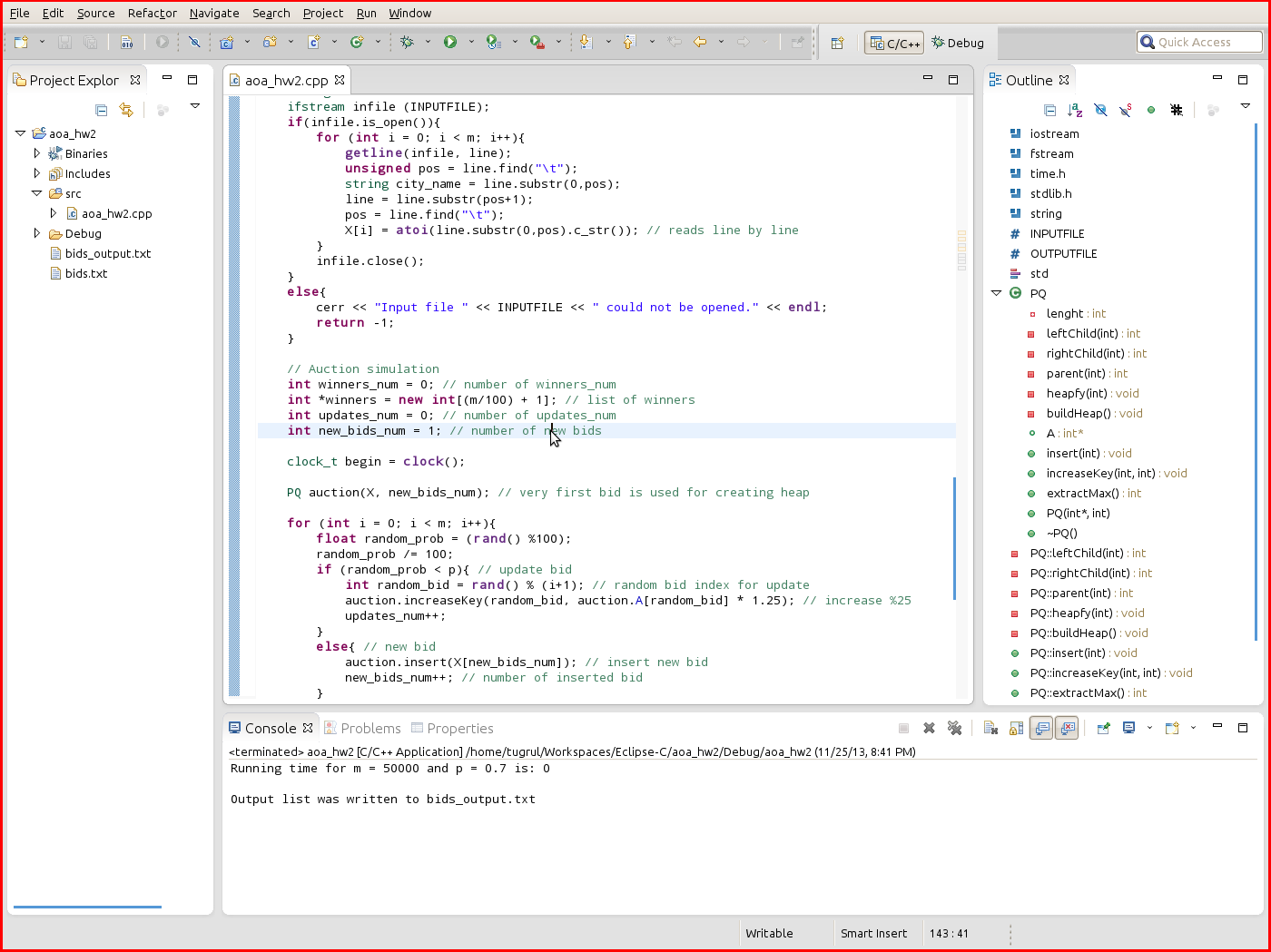
**STUDENT NUMBER: 040100117**

**1. Introduction**

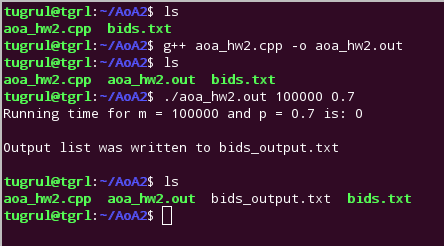
In this project, we build a priority queue (PQ). PQ is an abstract data type and we implement it using the heap data structure. In an online auction website, users can bid for an item and can increase their bids later. In addition, site manager periodically selects the highest bidder and sends him/her the item. After the submission, existing bids (except the winner) are retained and the users continue to bid and increase their bids so they can be the next highest bidder who will receive the next item.

**2. Development and Operating Environments**

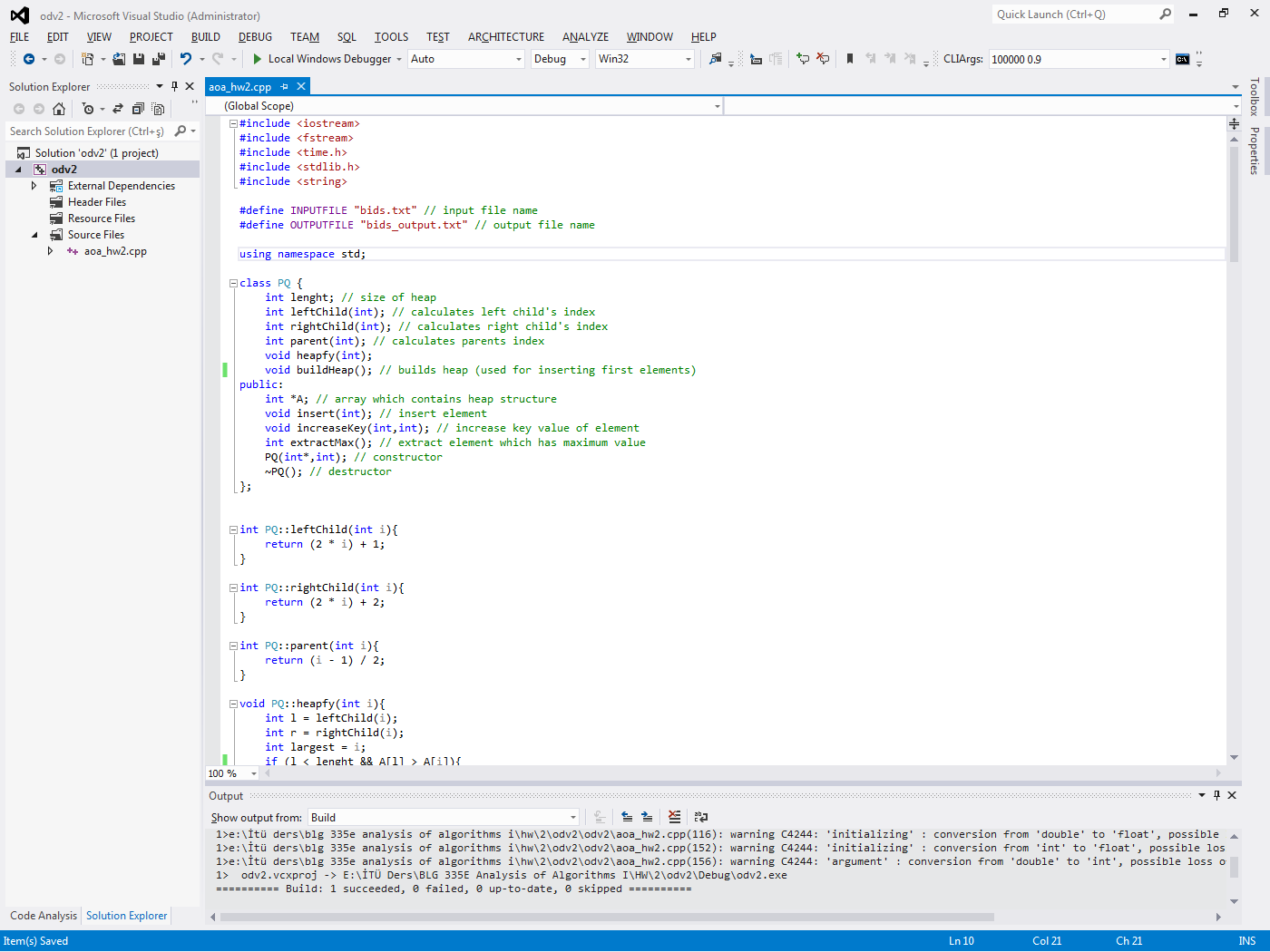
Eclipse for C++ integrated development environment has been used to write the source code in Ubuntu 12.04 operation system and GNU g++ compiler has been used for compiling under Ubuntu 12.04 operation system.

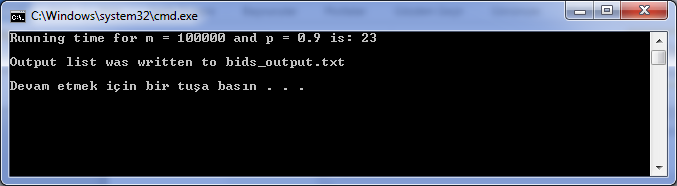


The program built and compiled without any warning or error under g++. Finally the program is executed. Sample outcome is below:



Due to precision of running time, simulation calculations are carried out on Windows 7 operation system and Microsoft Visual Studio 2012 development environment.





Sample output file is:

Running time for m = 100000 and p = 0.9 is: 23

Number of new bids: 10039

Number of updates: 89962

Auction winners are:

2288

2931

3032

2812

2752

2918

2635

2376

2478

…

**3. Data Structures and Variables**

In this project heap data structures is used for implementing priority queue abstract class. Methods of PQ class are explained in analysis section of this document.

* **#define INPUTFILE "bids.txt"**

Input file name

* **#define OUTPUTFILE "bids\_output.txt"**

Output file name

* **int lenght**

Size of heap array

* **int \*A**

Array which contains heap structure

* **int \*winners**

Array which contains winners

**4. Analysis**

**1)**

* **int leftChild(int)**

Returns left child’s index of an element

Running time is **Θ(1)**

* **int rightChild(int)**

Returns right child’s index of an element

Running time is **Θ(1)**

* **int parent(int)**

Returns parent’s index of an element

Running time is **Θ(1)**

* **void heapfy(int)**

Maintains the heap property.

The subtrees of children of our current node have at most 2m/3. The running time of heapfy is:

T(m) T(2m/3) + Θ(1)

According to Master method Case 2;

Running time is T(m) = **O(lg m)**

* **void buildHeap()**

Use heapfy in a bottom-up manner to convert an array A[1..n] into a heap.

Running time is **O(m)**

* **void insert(int)**

New user bids can be added to the heap structure.

Running time is T(m) = **O(lg m)**

* **void increaseKey(int,int)**

Value of an element (bid) can be updated by its index in heap.

Running time is T(m) = **O(lg m)**

* **int extractMax()**

Maximum node (bid) can be removed from the heap.

Running time is T(m) = **O(lg m)**

Main algorithm:

|  |  |
| --- | --- |
| for (int i = 0; i < m; i++) | **O(m)** |
| if (random\_prob < p)  int random\_bid = rand() % (i+1);  auction.increaseKey(random\_bid, auction.A[random\_bid] \* 1.25);  updates\_num++; | **O(1)** |
| **O(1)** |
| **O(lg m)** |
| **O(1)** |
| else  auction.insert(X[new\_bids\_num]);  new\_bids\_num++; | **O(1)** |
| **O(lg m)** |
| **O(1)** |
| if ((i + 1) % 100 == 0)  winners[winners\_num] = auction.extractMax();  winners\_num++; | **O(1)** |
| **O(lg m)** |
| **O(1)** |

Worst case running time of algorithm:

O(m)\*[O(1) + O(lg m) + O(1)] = O(m) \* O(lg m) = **O(mlg m)**

But this is the worst case scenario and it does not give us a tight running time. Heap-size will be maximum when the all operations are insert actions. But action is insert until 1-p probability is occurs, so that heap-size will be **m\*(1-p)** at the end of the execution. Algorithm starts with heap-size = 1, and it increases every insert operation. At the end of the execution, heap-size will be like **m\*(1-p),** but in execution phase we cannot know the exact heap-size. We know that average case of running time will be much shorter than the worst case running time. Mostly the algorithm runs on quickly than O(mlg m).

**2)**

A table and a graph demonstrate the effect of the m choice on the running time. Table is consist of different values of m between 1000 and 100000 for a constant p = 0.2

|  |  |
| --- | --- |
| **p=0.2** | |
| **m** | **Runing**  **Time** |
| 1000 | 0 |
| 3000 | 1 |
| 6000 | 2 |
| 10000 | 3 |
| 15000 | 4 |
| 20000 | 6 |
| 30000 | 9 |
| 40000 | 12 |
| 50000 | 15 |
| 60000 | 18 |
| 70000 | 21 |
| 80000 | 24 |
| 90000 | 27 |
| 100000 | 30 |

**3)**

As can be seen in the graph running time is effected linearly from m values. We can guess from graph that the running time of algorithm is something like;

T(m) = **O(m)**

Our theoretical worst case running time was **O(m lg m)** and theoretical average running time was something bellow the worst case, it may be **O(m)**. As can be understood from actual results, the algorithm runs on average running time which is **O(m)**. In this algorithm theoretical worst case running time is not decisive.

**4)**

A table and a graph demonstrate effect of the p choice on the running time. Table is consist of different values of p between 0.1 and 0.9 for a three different values of m = 10000, 50000 and 100000 for observation of choosing appropriate m value.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **m** | | |
| **10000** | **50000** | **100000** |
| **p** | **0,1** | 3 | 15 | 31 |
| **0,2** | 3 | 15 | 30 |
| **0,3** | 3 | 14 | 29 |
| **0,4** | 3 | 14 | 28 |
| **0,5** | 3 | 13 | 26 |
| **0,6** | 2 | 13 | 24 |
| **0,7** | 2 | 12 | 24 |
| **0,8** | 2 | 12 | 23 |
| **0,9** | 2 | 12 | 22 |

**5)**

As can be seen in the graph p value effects running time slightly especially bigger m values. p is the probability of update bid action and p-1 is the probability of new bid action. Update bid action is provided by *increaseKey* algorithm in heap structure and new bid action is provided by *insert* algorithm in heap structure. Both *insert* and *increaseKey* algorithm has running time of **O(lg m)**. Main algorithm must run one of the *insert* and *increaseKey* in one loop in whatever possibility is. Although they are the same bound of running time **O(lg m)**, their coefficient on running times are different. Coefficient of *increaseKey* is smaller than the coefficient of *insert.* So that when p values decreases, probability of *insert* algorithmincreases and *insert* has higher running time coefficient than the *increaseKey* , running time of main algorithm is increases.

*Insert* algorithm is always calls heapfy algorithm from last leaf of heap so that almost every time heapfy is running on worst case, but *increaseKey* algorithm takes random index to increase value mostly runs on average running time. For this reason *Insert* algorithm *is slower than increaseKey* algorithm.

**5. Conclusion**

In this homework, I have become more familiar with the concept of heap and analysis of the algorithm. I had the chance to intensify my knowledge about instructing good and efficient algorithms.