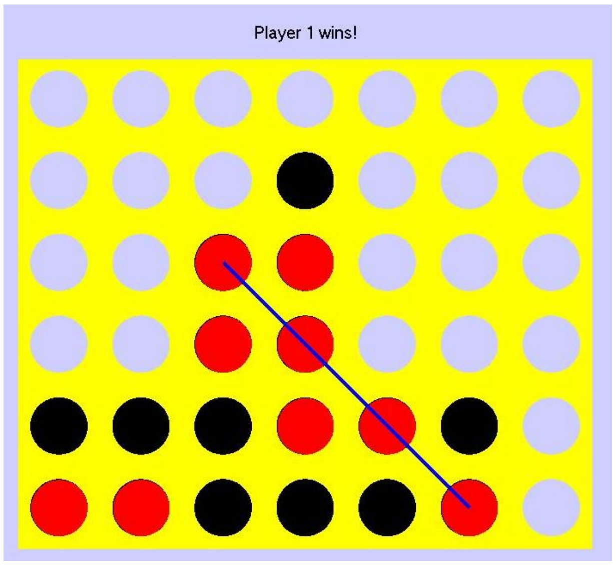
Assignment 3

Due by 20.6 (noon)

In this assignment, you are asked to implement the **Connect4** game where the user plays against the computer.

Background

Connect4 (aka **four in a row**) is a two-player game in which players take turns dropping colored discs into a seven-column board. The pieces fall straight down, occupying the next available space within the column. Each column contains 6 cells – so the entire board has 6 lines. The goal of the game is to connect four consecutive discs of the same color. The sequence can be either horizontal, vertical or diagonal. The player who succeeds to do so (clearly with his own colored discs) is the winner.



Goal

In this assignment, you will implement the connect4 game. You are asked to develop a command-line program which consists of two parts; initialization and the game itself.

In the initialization part, the user, among other things that will be described later, chooses the difficulty level of the computer.

After the initialization is complete, the game begins with the user turn. In each turn of the user, you should print the game status in a predefined format (explained later), and awaits for the user. The game continues in this manner until either a winner is announced or the board gets full of discs in which the game is tied with no winners.

Initialization

The game starts by asking the user to enter the difficulty level. The program prints the following message and waits for the user's input.

"Please enter the difficulty level between [1-7]:\n"

The user must then enter a number representing the desired difficulty, which is a number between [1-7] (larger number means more difficult game). As we describe later on, this parameter defines the depth of the minimax algorithm.

If the user enters an invalid number then the program prints an error message and lets the user try again by printing the same message as above. This phase finishes only upon providing valid level. You can find the corresponding error message in the error messages section.

The user may exit the game any time by invoking the command 'quit'.

The game

The game starts by creating an empty game board. The user goes first, followed by a computer move and vice versa. In a user turn, he needs to decide whether to drop a disk or undo his previous move.

If the user enters a valid move, then a disk with the user color is placed on the board.

If the user undoes a previous move, then the last two disks to be put on the board will be cancelled.

The game board status is printed before every user turn, followed by an appropriate message which will be defined shortly.

Furthermore, the user may decide to quit the game or restart the game by invoking the appropriate commands. For more information, please refer to the following sub-sections.

Game status - printing format

We will use the following format for printing the status. The status will contain 8-lines, where each consists of 17 characters.

Each line starts and ends with the '|' character. In between, there are 15 characters (except for the last two lines). 7 of the characters will represent the status of the cell: 'X' means user disc, 'O' means computer disc and ' ' (space) means that there is no disc yet in this cell. Each two neighbor characters as described above will be separated by space (‘ ‘). See the example below.

There are 6 such lines as the game contains 6 lines. The 7th line should contain a series of 17 dashes ‘-‘.

The bottom (8th) line contains the column numbers (1 through 7) in the columns corresponding to disc positions, exactly as shown below.

Next is an illustration of two examples the beginning of a new game (left), and end of a game (right):

```
|               |                  |               |
|               |                  |               |
|               |                  |           X   |
|               |                  |   O     X O X |
|               |                  | O X O X O X X |
|               |                  | O O X X O X O |
-----------------                  -----------------
  1 2 3 4 5 6 7                      1 2 3 4 5 6 7
```

Recall that the game board is printed in each round. If there is no winner, the program prints the board followed by the message:
“Please make the next move:\n”.

If the user wins then the game board is printed followed by the message:
“Game over: you win\nPlease enter ‘quit’ to exit or ‘restart’ to start a new game!\n”
If the computer wins, the game board is printed followed by the message:
“Game over: computer wins\nPlease enter ‘quit’ to exit or ‘restart’ to start a new game!\n”

If the board is full and no one wins, the game board is printed followed by the message:

"Game over: it’s a tie\nPlease enter ‘quit’ to exit or ‘restart’ to start a new game!\n”

Notice that in any case, the program then waits for a command. The only commands allowed when the game is over are, ‘restart’, ‘quit’ and ‘undo_move’.

Commands

The program repeatedly gets commands from the user as follow:

1. The command includes at least one non-whitespace character.

2. The command and its parameters are pairwise separated by one or more white spaces (not including '\n')..
3. You may assume that a command is at most 1024 characters long.

Use $gets(str)$, where str is a pre-allocated $char *$ variable, to read the commands..

You should support the following commands:

suggest_move

This command executes the minimax algorithm for the **user** and proposes the best move. After executing the minimax algorithm (described below), the program prints the following message to the user:

```
"Suggested move: drop a disc to column X\n"
```

Where X is the column number suggested by the minimax algorithm (X is 1-based number).

Note: This command doesn't perform any move.

undo_move

This command undoes the two moves previously played, i.e. the moves played by the user and the computer.

```
"Remove disc: remove computer's disc at column X\n"
"Remove disc: remove user's disc at column Y\n"
```

Where X and Y are the column numbers played by the user and the computer in the previous turn, respectively. The program then prints the modified board to the user.

add_disc $< col\_num >$

This command is responsible for placing a disc in the specified column for the user. The computer move computation follows. Next the program prints the following message to the user:

```
"Computer move: add disc to column (xxx)\n".
```

Finally, the program then prints the game board to the user (as explained in the subsection 'Game status - printing format').

quit

The program prints:
"Exiting…\n", frees the memory and terminates.

restart_game

This command restarts the game by clearing the board and starting a new game. The game starts over from the initialization part. After invoking this command the program prints the following message:

```
"Game restarted!\n".
```

Special Remarks:

- *Please note that the columns are 1-based, i.e the first column is indexed 1. Thus whenever there is a message containing the column number, it must be 1-based.*

- **PLEASE CAREFULLY REFER TO THE ERROR HANDLING SECTION :** In this section you find all error messages while executing the program. When an error occurs, the game board will not be printed and only the relevant error message will be shown.

The Undo move - Revisited

As stated in previous sections, the user may undo moves previously executed. This means that the discs that are put in the previous rounds may be removed by invoking the undo_move many times.

Although the number of moves are at most 6×7, you should allow the user to undo only the previous 20 moves made so far. That is, the user is allowed to invoke the command only 10 times, since each time the command removes one disc for each player. Notice that we only count valid moves. For example, if there were 10 valid moves followed by 1000 invalid moves then invoking the undo_move command will remove disks placed by the 10 valid moves.

If the user invokes the undo_move command more than 10 times in a row, the program will print an error (see Error handling section). Further, if there are less than 10 valid moves made so far, the user will only be allowed to invoke the undo_move as many times as the number of valid moves.

For example, check the following scenario:

1- The user puts a disk on column 2.
2- The computer puts a disk on column 6.
3- The user invokes undo_move. The discs are removed so that the board is empty again.
4- The user invokes 17 valid moves denoted by $m_1, .., m_{17}$.
5- .The user invokes undo move 2 times (recall the limit of 10 such actions). Moves $m_{17}\ and\ m_{16}$ are cancelled, in that order.
6- The user invokes 3 valid moves, let m_{18}, m_{19}, m_{20} be these moves (in the order of execution). Notice that the moves recorded by the program are $m_1, .., m_{14}, m_{15}, m_{18}, m_{19}, m_{20}$ (that is m_{17} and m_{16} were cancelled).
7- The user invokes undo_move 10 times in row. The remaining moves are $m_1, .., m_8$.
8- Invoking undo_move command before playing any move will always result in an error.

NOTE:

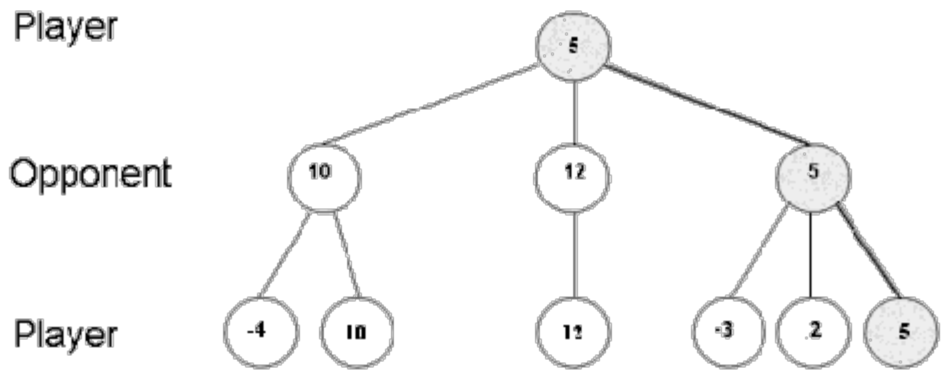
- If the game is over, the user may still invoke the undo_command.
- If the user starts a new game, then moves from the previous game cannot be restored.

Minimax algorithm

One of the popular AI (Artificial Intelligence) algorithms for two-player games is called Minimax. **Minimax** is a recursive algorithm for choosing the next move in a 2-player game (but also in an n-player game which is not of interest in this work). The main idea is as follows. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves.

Following this idea, the computer calculates its possible moves, gives a score to each move according to the game state it reaches and possible future states. The decision is done with a scoring function that we illustrate below. It calculates future moves from each such move recursively, along with a score for each resulting state, and then picks the best move for the current turn and plays it.

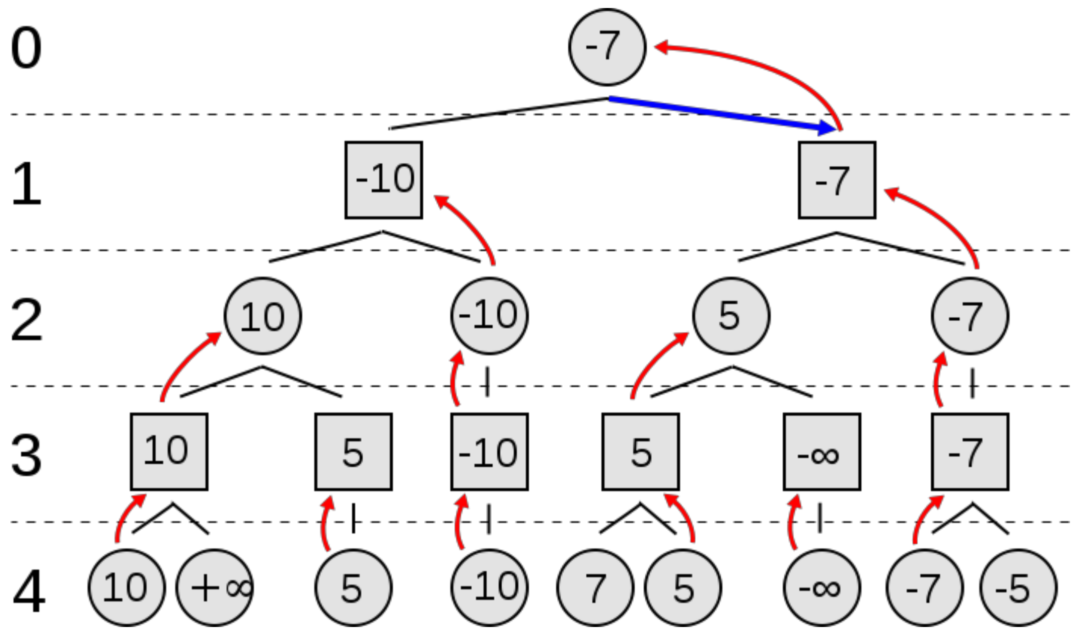
We can represent the results of the minimax algorithm as a tree where the vertices are game states and their score, and the edges are possible moves from each game state.



In the figure above, the game contains at most three legal moves for each turn, and the minimax algorithm calculates two steps ahead, with the score shown as the value of the node.

The minimax algorithm uses a “scoring function”. Given an input board, it produces a number. The higher the number is, the better the board for player A; the lower the number is, the better the board for player B.

With a scoring function and a way to find allowed moves for a particular board (game state), we can construct the tree illustrated below.



Each node represents a game state, with the root node representing the current state of the game. From the root we recurse down the tree, creating nodes that represent possible game states that can occur with each of the allowed moves. The process finishes when the specified depth is reached or we reached a final state (in other words, when the recurance depth has reached the input specified depth or the game is over).

For example, at the beginning of the game the root node represents a new game state for a blank board. It has seven edges (seven children) – each representing the move for player A when placing a disc in columns 1-7, connecting to a node representing the new game state (with a single disc at the bottom of the corresponding column). We then expand each state to its own 7 child nodes – each representing the move for player B when placing a disc in columns 1-7, for each of the previous states – for a total of 49 **new** nodes (game states), and so forth until reaching depth equal to the maximum depth that was specified at the initialize part (i.e, that is the difficulty level) or reaching a final state (i.e, a state where the game is over).

Important note: not all game states will have 7 children, as some game states have full columns.

The algorithm continues to build the tree until we reach the maximum depth level, and we then need to compute the scoring function. It follows that scores are computed for all the leaves of the tree. We apply a simple strategy for other nodes (consider the root as level 0):

- If the level corresponds to a move of player A (an even-numbered level), we set the maximum of the children scores to their parent (since player A wants to maximize the scores).
- If the level corresponds to a move of player B (an odd-numbered level), we set the minimum of the children scores to their parent (since player B wants to minimize the score).

This process compute scores as in the figure above. When the recursion computes all scores at level 1 (depth 1), the first level below the root, the final result is computed – and the best move is chosen according to the one that leads to a child with the optimal score.

Special remarks:

- **Tie-Breaker: if there are two or possible moves that leads to the same maximum score then you need to choose the move with the lowest column index.**
- When the MiniMax algorithm process the computer move, it is treated as player A. However, when the user inputs the command to suggest a move, the user is treated as player A.

Scoring Function

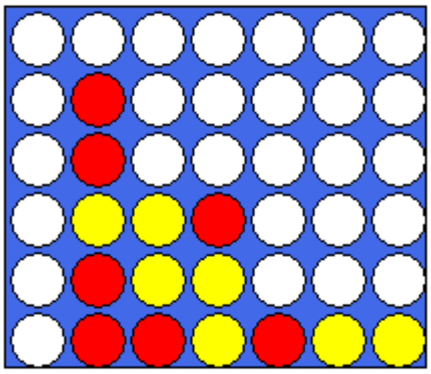
The scoring function relies on the spans of disks on the board.

To compute the score, we go over all possible spans of 4 discs on the board. For each span, we count the number of discs it contains for each player.

In each span there can be at most 4 discs. We count player A's discs as positive: $+1$, and player B's discs as negative: -1. This process gives us, for each span, a score number between -4 and $+4$. (Note that +4 and -4 actually means that there is a winner.)

We then aggregate the results in a an array of 7 elements that represents values from -3 to +3 . Each element of the array counts the number of times we encountered a span of that size. We can ignore spans of size 0. (Notice that we are actually calculating a histogram.)

For example, here is a game board along with its score table where the **red** player is A:



$[-4] = 0$	$[-3] = 1$	$[-2] = 3$	$[-1] = 17$
$[4] = 0$	$[3] = 0$	$[2] = 5$	$[1] = 12$

Once we have this table calculated for all possible spans, we can calculate the board's score.

If there's any $+4$ or -4 score (there is a winner),then the score is assigned a huge value (use INT_MAX and INT_MIN defined at limits.h).

In any other case, we'll calculate the score according to a constant weight vector.

Our weight vector is: $\{-5,-2,-1,1,2,5\}$

Each amount is multiplied by this vector. This means that we multiply the number of $[-3]$ spans we found by the value -5, $[-2]$ by the value -2, etc.

We sum up the 6 values that we get and that's the score for the given board (we ignored spans with the value 0).

For the above board the score is:

$$1\cdot(-5)+3\cdot(-2)+17\cdot(-1)+5\cdot 2+12\cdot 1=-6$$

The scoring function should take $O(n)$ time, where n is the size of the board.

Remarks:

- we emphasize that the above scoring function is only applied on a leaf node in the MiniMax tree.
- A detailed example can be found on moodle.

Error Handling

The program must print only **one** error message, even if more have occurred. Print the error with the smaller serial number (i.e., prefer 1 to 2, etc.).

1. Standard failures such as memory error ($malloc$, $scanf$, etc). The error message is `"Error: <function name> has failed"`. This is the **only** edge-case when you **must free** the memory allocated and exit the program.
   <u>Note:</u> you do not need to check if $printf$ succeeded. Those failures are considered rare and most programmers prefer a clean code to that check.

2. In the initializing part, if the user enters invalid MiniMax depth then the following error is printed.
   `"Error: invalid level (should be between 1 to 7\n"`.

3. If the user enters an invalid line command (i.e. that doesn't match any of the commands defined in previous sections) - the program prints to the standard input `"Error: invalid command\n".` An invalid command is either one of the following:
   - The command doesn't start with one of the operations we defined in the previous sections.
   - The command does not match the syntax we defined.
4. In $\boldsymbol{add\_disc}\ < col\_num >$: if <col_num> isn't between 1 and 7, the program prints `"Error: column number must be in range 1-7\n"`.
   Examples:
   - add_disc abc12, add_disc 10.11, add_disc 7.0, add_disc -10.

5. In $\boldsymbol{add\_disc}\ < col\_num >$: If <col_num> refers to a full column (already contains 6 discs), the program prints `"Error: column X is full\n"`.

6. In ***undo_move***: If undo command cannot be invoked (refer to the undo move section), then the program prints the following error:
   "Error: cannot undo previous move!\n"

7. In case the game is over (the user or the computer won on the last move),the following commands are not allowed: ***add_disc***, ***suggest_move*****.** In case the user enters any of these commands, the program prints: `"Error: the game is over\n"`.
   NOTE: The only commands allowed are ***undo_move*****, quit** and **restart.**

Remark: the only cases were the program terminates is either if error 1 occurred or the user enters the quit command. We emphasize that you must free all memory resources upon termination.

Implementation overview

In this assignment you need to implement different modules which will be combined to form the entire program.

Notice that these modules were carefully picked and described so it would ease the implementation. If you think that some of the functions are useless, you are still required to implement them, since your code will be later integrated with an already implemented graphical user interface (GUI) of the connect 4 game. Furthermore, these modules can be extended. If you add new functions, you have to provide a documentation of these functions as given in the module files.

SPFIARParser

As the user inputs lines that represents commands, you need to read and interpret them.

This should be the responsibility of the module SPFIARParser which consists of many useful functions that we believe are relevant to this assignment.

Please carefully review the header SPFIARParser.h that is in the assignment zip file, an overview of the module along with a detailed behavior of each function is provided there. Your implementation must agree with the behavior provided in the documentation.

Note: it is very recommended to use the standard function ***strtok***, which can be found in the header file ***string.h***. This function is similar to the split method in java. Use " \t\r\n" as a delimiter string, more information can be found online

SPArrayList

As specified in the specification of the assignment, we need to be able to undo previous moves. For this reason, we will implement a data-structure that saves the last 10 moves of each player. For this reason we will implement the following data structure.

The data structure implements a fixed-size array list. Just like in java, our data structure will support adding, removing and accessing the list elements. However, our list will have a maximum capacity such that if it's reached then no further elements can be added.

The elements of the list will be integers that represent the column numbers of the previous move. Notice that you need enough space to record the last 20 moves, 10 for each player. Please refer to the header file provided in the assignment zip file.

SPFIARGame

This module is an encapsulation of the game board. It will be used to store and manipulate the game status. It provides many functionalities such as checking if a move is valid, setting a move, removing a disc from a specified column and so on.

Please carefully review the header SPFIARGame.h for more information.

SPMiniMax

This module implements the MiniMax algorithm. Notice that the user of this module need not to worry about the algorithm behind the module. Thus it should only support basic functions such as creation of the MiniMaxTree, and suggesting the next move. All algorithmic ideas behind the implementations will be placed on the file SPMiniMaxNode.

SPMiniMaxNode

This module represent a node in the MiniMax tree. The node contains all the data needed to implement the tree, plus the module will have to support functions that are useful to you when you implement the functions in the SPMiniMax module.

It is recommended to implement recursion functions inside this module, and to call them from SPMiniMax module. The module header is provided in the assignment file, and it needs to be extended to match your implementation. Notice that any function you provide must be well-document as provided to you in the other modules header files.

main.c

main() should be the only function in this module.. Any auxiliary function that you think might be useful should reside at the SPMainAux module.

SPMainAux

All auxiliary functions are placed inside this module. Notice that you need to place only functions that don't fit to any of the previous modules.

Testing

It is recommended to test every module separately before writing your main function. Every module you implement must have the same behavior as stated in the documentation of that module.

To test every module, you need to implement small programs that are called unit tests. Every unit test should cover as much edge cases as possible, this will ease the debugging phase and should make the developing process faster. You're provided with basic unit tests for the SPFiarGame and SPArrayList modules. The unit tests uses the header file unit_test_util.h which provides useful macros that can be used during testing. Also notice that these unit tests don't cover all cases and should be extended.

Special Notes:

- Every unit test you make sure end with the suffix UnitTest.c. For example if you want to make a unit test for SPFIARParser then your unit test source file name should be SPFIARParserUnitTest.c

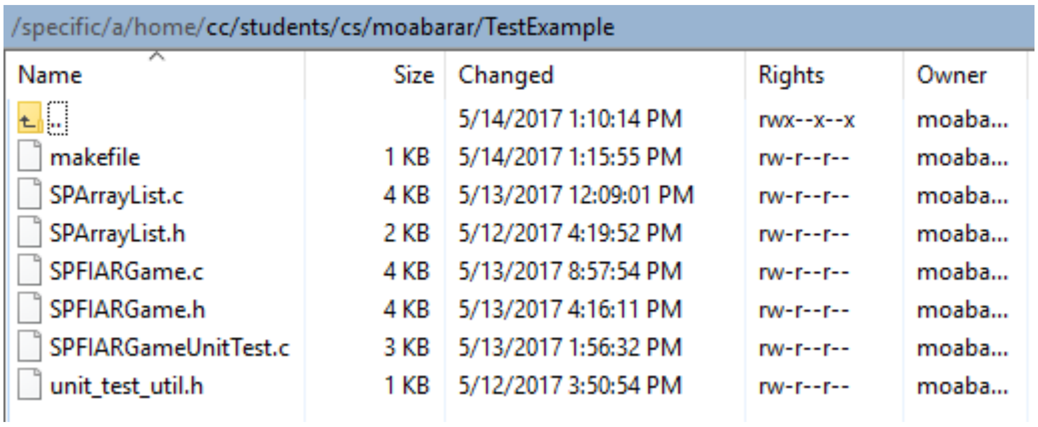
Unit Testing on Nova

All your modules should manage the memory resources properly, thus you should not have memory leaks during the execution of your unit tests.

To check that if your module has any memory errors, you need to run valgrind on your unit test as well (more on valgrind can be found on moodle website). Please make sure your unit test is contains no memory errors, for example, if you create an instant of an ArrayList you need to make sure it is destroyed by the end of use (refer to the provided unit tests as an example).

Here an example on how to run SPFIARGameUnit tests on Nova:

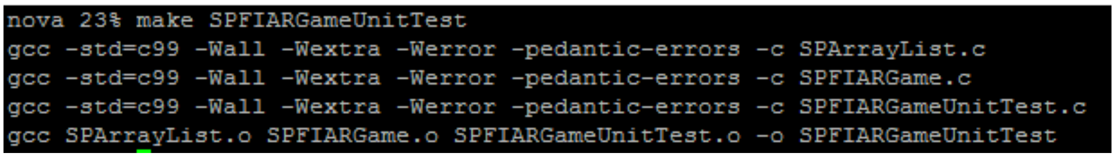
First you need to copy all relevant files to nova, such that all files should be under the same directory (in this example you need to have SPFIARGame.h, SPFIARGame.c, SPArrayList.h, SPArrayList.c, SPFIARGameUnitTest.c and unit_test_util.h). You will also need to put the makefile provided in the assignment zip file to the same directory as will.



/specific/a/home/cc/students/cs/moabarar/TestExample

Name	Size	Changed	Rights	Owner
..		5/14/2017 1:10:14 PM	rwx--x--x	moaba...
makefile	1 KB	5/14/2017 1:15:55 PM	rw-r--r--	moaba...
SPArrayList.c	4 KB	5/13/2017 12:09:01 PM	rw-r--r--	moaba...
SPArrayList.h	2 KB	5/12/2017 4:19:52 PM	rw-r--r--	moaba...
SPFIARGame.c	4 KB	5/13/2017 8:57:54 PM	rw-r--r--	moaba...
SPFIARGame.h	4 KB	5/13/2017 4:16:11 PM	rw-r--r--	moaba...
SPFIARGameUnitTest.c	3 KB	5/13/2017 1:56:32 PM	rw-r--r--	moaba...
unit_test_util.h	1 KB	5/12/2017 3:50:54 PM	rw-r--r--	moaba...

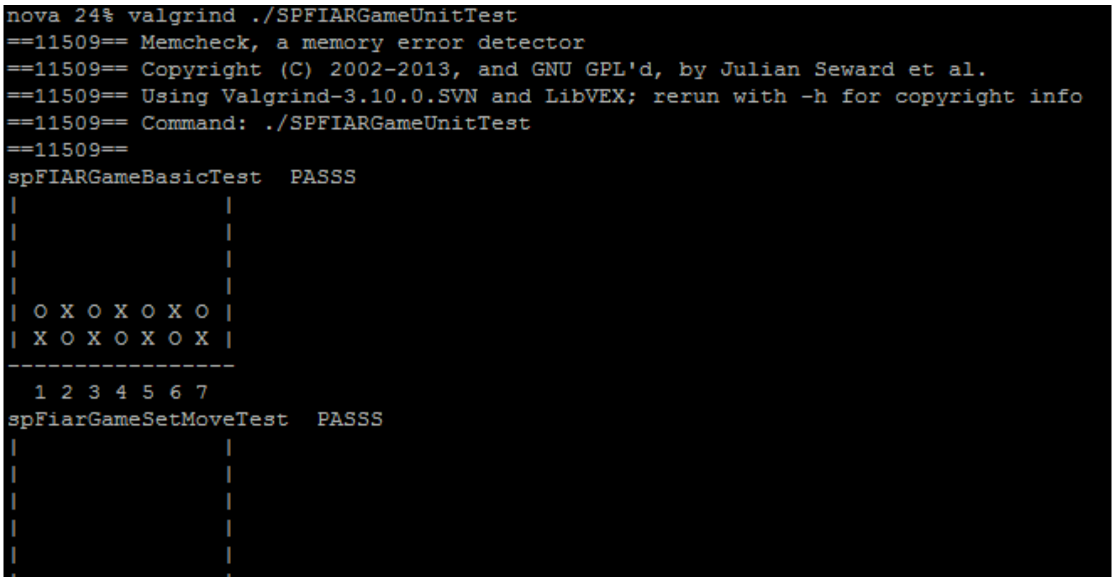
Now you need to go to the desired directory (TestExample in the above example) and compile the unit test using the makefile provided to you in the assignment zip file. In order to compile a specific unit test, you need to invoke the specified target of that unit test. The unit test target name is the same name of the module with the suffix UnitTest. For the above example you need to invoke: ***make SPFIARGameUnitTest*.**



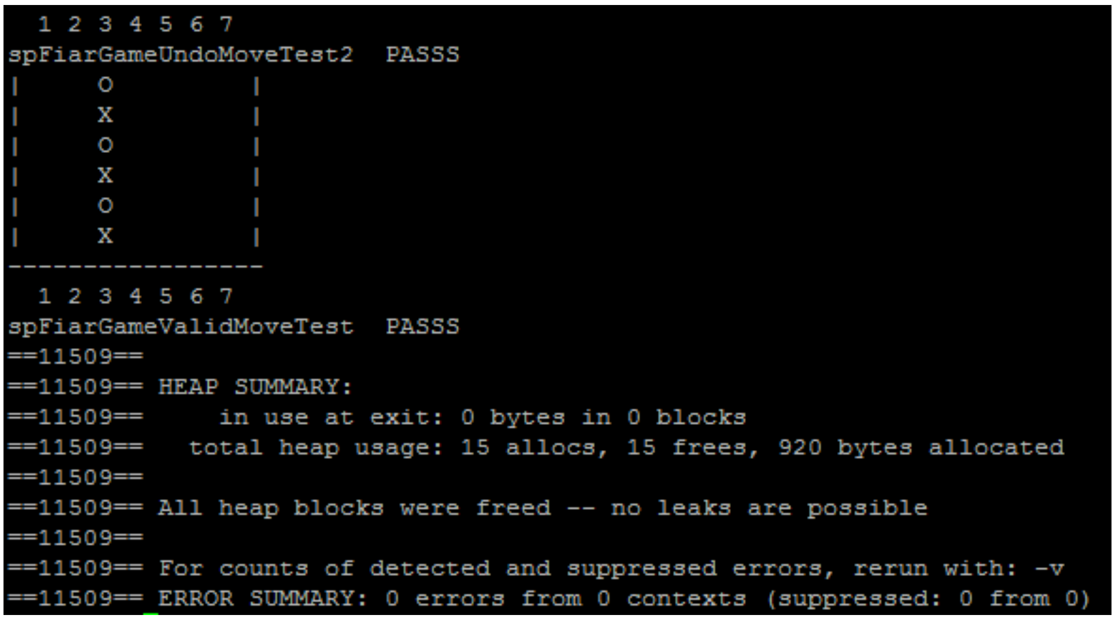
```
nova 23% make SPFIARGameUnitTest
gcc -std=c99 -Wall -Wextra -Werror -pedantic-errors -c SPArrayList.c
gcc -std=c99 -Wall -Wextra -Werror -pedantic-errors -c SPFIARGame.c
gcc -std=c99 -Wall -Wextra -Werror -pedantic-errors -c SPFIARGameUnitTest.c
gcc SPArrayList.o SPFIARGame.o SPFIARGameUnitTest.o -o SPFIARGameUnitTest
```

After the compilation process is done, you an executable with the name of the module and the suffix UnitTest is created (in our example the binary name is SPFIARGameUnitTest).

You can run the binary with valgrind in order to check for memory errors, as given in the example below (the picture was cropped by half for space restrictions):



```
nova 24% valgrind ./SPFIARGameUnitTest
==11509== Memcheck, a memory error detector
==11509== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==11509== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==11509== Command: ./SPFIARGameUnitTest
==11509==
spFIARGameBasicTest  PASSS
|               |
|               |
|               |
|               |
| O X O X O X O |
| X O X O X O X |
-----------------
  1 2 3 4 5 6 7
spFiarGameSetMoveTest  PASSS
|               |
|               |
|               |
|               |
|               |
```



```
  1 2 3 4 5 6 7
spFiarGameUndoMoveTest2  PASSS
|     O         |
|     X         |
|     O         |
|     X         |
|     O         |
|     X         |
-----------------
  1 2 3 4 5 6 7
spFiarGameValidMoveTest  PASSS
==11509==
==11509== HEAP SUMMARY:
==11509==     in use at exit: 0 bytes in 0 blocks
==11509==   total heap usage: 15 allocs, 15 frees, 920 bytes allocated
==11509==
==11509== All heap blocks were freed -- no leaks are possible
==11509==
==11509== For counts of detected and suppressed errors, rerun with: -v
==11509== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note that you can also run valgrind with different flags as given in the valgrind manual.

Special Notes:

- There are separate targets for each unit test in the makefile file. In order to compile any module, you need to invoke **"make <module_name>UnitTest"**, where <module_name> is replaced with the actual name of the module.
- All files related to a specific modules must be in the same directory along with the header file unit_test_util.h. For example in SPFIARGame module, we used the SPArrayList to store the previous moves made in the game, thus both SPArrayList.c and SPArrayList.h were included in directory.
- A list of the related files to a module appears below:
  - **SPArrayList:** SPArrayList.h, SPArrayList.c SPArrayListUnitTest.c and unit_test_util.h
  - **SPFIARGame:** SPFIARGame.h, SPFIARGame.c, SPArrayList.h, SPArrayList.c SPFIARGameUnitTest.c and unit_test_util.h
  - **SPFIARParser:** SPFIARParser.h, SPFIARParser.c, SPFIARParser UnitTest.c and unit_test_util.h
  - **SPMiniMaxNode.h:** SPMiniMaxNode.c, SPMiniMaxNode.h, SPFIARGame.h, SPFIARGame.h, SPFIARGame.c, SPArrayList.h, SPArrayList.c, SPFIARMiniMaxNodeUnitTest.c and unit_test_util.h
  - **SPMiniMax.h:** SPMiniMax.c, SPMiniMax.h SPMiniMaxNode.h, SPMiniMaxNode.h, SPFIARGame.h, SPFIARGame.h, SPFIARGame.c, SPArrayList.h, SPArrayList.c, SPFIARMiniMaxUnitTest.c and unit_test_util.h

Putting it all together

You can compile your main function along with all parts of your program by simply invoking **"make".** The executable filename is SPFIAR.

"make clean" will clean previous compilations.

Closing notes and special remarks

- It is recommended to start working on the following modules (as appears in order):
  - SPFIARParser
  - SPArrayList
  - SPFIARGame
  - SPMiniMaxNode
  - SPMiniMax

  Once you are finished implementing these modules, you can start implementing the main function.
- You need to document every function you add to any of the modules.

Submission guidelines

At least one of the students should submit a zipped file with the name **“id1_id2_assignment3.zip”** where id1 and id2 are the ids of the partners. The zip file should include the following files (without subdirectories):

- **Source files:** SPFIARGame.c, SPFIARParser.c, SPArrayList.c, SPMiniMax.c, SPMiniMaxNode.c, SPMainAux.h and main.c (**note that the unit tests should not be included)**
- **Header files:** SPFIARGame.h, SPFIARParser.h, SPArrayList.h, SPMiniMax.h, SPMiniMaxNode.h and SPMainAux.h

> Good Luck