Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Spring 2023

Homework 3 –Class Design for Futuristic Backgammon Game using Stack and Doubly Linked List

Due: April 12, 2023, Wednesday, 21:00

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

You can NOT collaborate with your friends and discuss solutions. You have to write down the code on your own. Plagiarism and homework trading will not be tolerated!

Introduction

For this homework, you will build a backgammon-like game class that utilizes a *stack* of game pieces and a doubly linked list of **`slot`** *structs*. In other words, the **`slot`**s will keep user pieces that are represented as *stack*s. At the beginning of the program, the current state of the game board will be read as a string and then the board will be generated as a doubly linked list of **`slot`**s. The game implementation uses a class called **`Board`**. The member functions of **`Board`** class have been used to implement the game mechanics given in the main program (main program has been provided to you). What you will do in the homework is to implement the **`Board`** class that has member functions for different ways of slot creation, slot deletion, checking whether any move is possible, moving a piece from one slot to another, to name a few (see below for more details). All these will be implemented as member functions. Details are explained in the following sections.

The Program Flow and the Game Rules

In this section, we explain how the main program works. Actually, this main program has been implemented by us and given to you. However, in order to understand the **`Board`** class' member functions well, we explain the main and the game mechanics here.

The game is basically a version of famous backgammon game. There are variable number of slots on the board. Each slot can either be empty or claimed by one of the user. If claimed by a user, a particular slot can contain only one type of piece (**`x`** or **`o`**). The maximum number of pieces that

can be put on a slot is 4; however, when the number of pieces on a slot reaches 4, the slot is entirely destroyed; thus we never see a slot with 4 pieces on it. The game is played in rounds and at each round both players play in turn. A user plays first by throwing a die that returns a random number between and including 1 and 4. Then the user plays one of its pieces on a selected claimed slot (the one on the top of the slot if there are multiple pieces on the selected slot), die amount of slots toward left or right. However, the target slot must either be claimed by the same user or must be empty for a valid move. If the target slot has been claimed by the other player or beyond the boundaries of the board, then such a move is invalid. That means, you cannot throw a piece out of board and you cannot place a piece on top of your opponent's piece. During the game, it'd be possible to have situations that the user cannot play any of its pieces; in such a case, the user should create a new slot (to the beginning or to the end, depending on its choice) and put one piece there.

As you can see, the number of pieces at the original setup remains the same if a piece is moved from one slot to another. However, when a particular slot reaches 4 same type pieces, since we destroy that slot, number of pieces is reduced. Moreover, when a move is not possible, one piece is added on a new slot and the number of pieces is increased. The strategy of the game is to end up with smallest number of pieces at the end of the game. The player with smallest number of pieces wins the game (of course tie is also possible). You may check out the sample runs to better understand the game logic.

Now let us go over how the main program has been implemented in the provided cpp file. First, the program asks for a string input representing the initial setup of the game board. The user will input a string consisting of separators, game characters (**`x`** or **`o`**) and the number of game characters per slot. The program checks if the string is valid according to the following criteria.

1. Each slot is represented using a digit and a character. The digit is the number of pieces in this slot and the character is the player's piece (**`x`** or **`o`**). For example, **`2x 1o`**
2. After a slot, there should be a separator character, which is **`/`**, if this slot is not the last slot.
3. If there are more than one separator character after a slot, each extra one represent an empty slot with no player piece. For example, **`3x///1o`** means one slot with three x's, two empty slots and one slot with one o.
4. Rule 3 above is valid for the separators in the middle of the string. If there are separators at the beginning or at the end, then this means there are that amount of empty slots. For example, **`/3x///1o//`** means one empty slot, one slot with three x's, two empty slots, one slot with one o and two empty slots.
5. Number of x or o's cannot be smaller than 1 and cannot be more than 3. (e.g. **`0x`**, **`4o`**, **`12x`** are not allowed).
6. No mixing of characters is allowed (e.g. **`1x2o`** is not allowed).

If the string is not a valid representation of a board, the program will show an error message and ask for a correct input from the user. After input validity checks, the program will parse the string to fill in the board object. To do so, **`Board`** class' **`createSlotEnd`** and **`createEmptySlotEnd`** member functions are used. Once the board object is successfully filled, the program will print the board in the following way using the member function **`printBoard`** (the base will be printed, but it is not stored as part of the **`slot`** structure). Two examples are given in Figure 1 and Figure 2

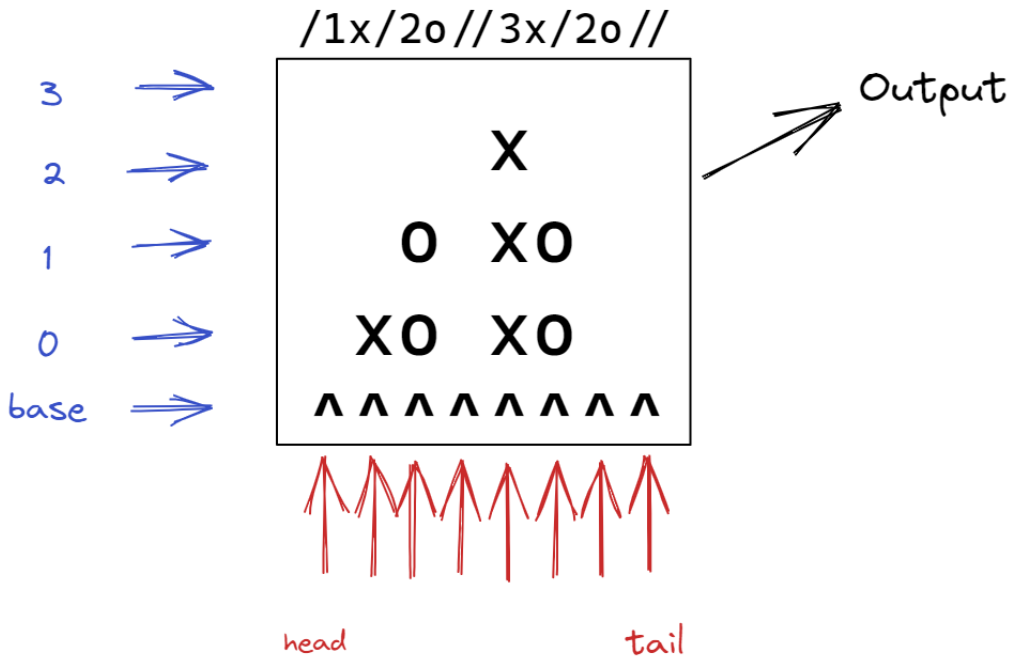


Figure 1: Example format of the printBoard function

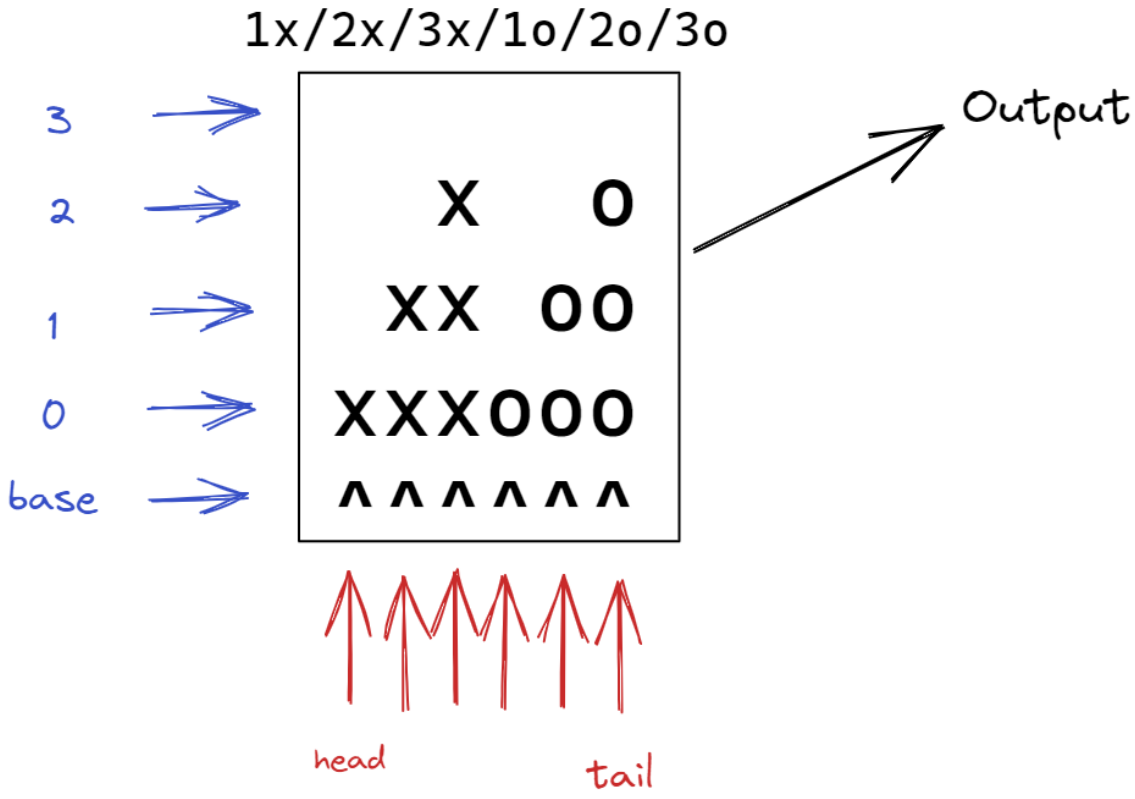


Figure 2: Second example format of printBoard function

The printing function will traverse the slots and print their contents. A slot will be signified by a caret '^' character at its base, this will make it easier to identify the empty slots. However, you will not store the caret character in the slot structure. For more examples of the display outputs, please see the sample runs.

Then, the program will ask for the *number of rounds* that the game will be played. The user will enter an integer for the round count. Following that, the user will be asked to enter a Pseudo Random Number Generator (PRNG) *seed*, which will be used to generate a sequence of random numbers for the die outcomes during the game. This random number generation function is a free function and given in the main program file to you.

At each round of the game, the players will play in turns (first **`x`**, then **`o`**). At each turn, the program displays the round number, and player character. Then die is rolled and the program calls **`noMove`** member function to check if the player has any possible valid move(s) with the given die result. There are two possible outcomes of this **`noMove`** member function:

- If there are no valid moves, then the program asks the user to append a slot with its game piece (one piece only) at the beginning (using **`createSlotBegin`** member function) or at the end (using **`createSlotEnd`** member function) of the board.
- If there is at least one valid move, the program asks the index where the player has a piece and whether to go die times forward (right) or backward (left). Then, the member function **`validMove`** will be called in order to check the validity of the requested move of the player. If the move is valid, then this function will return 0. However, there may be cases where the user will try to move other user's piece, go out of bounds of the board, or move its piece to an unavailable location. Here are the error possibilities to check in **`validMove`** function.

    1. Entered slot index is not within bounds.
    2. Target slot index not within bounds.
    3. Target slot index is not empty ~~or~~ and does not belong to the player.
    4. Entered slot index does not belong to the player.

    These error messages have already been incorporated in the main function provided. You are asked to implement the function **`validMove`** which will return an integer stating the outcome (0 or error code). If the move is not valid, the program will ask another entry until it is correct. Existence of a valid move is guaranteed since we check this before by calling **`noMove`** function.

Once the valid move is entered, the piece is moved using the public member function **`movePiece`**, to target slot.

After the move, the program checks if the target slot has four pieces, meaning the slot is full. In this case, we delete that slot. To check if the slot is full, you will use a public member function **`targetSlotFull`**. In order to delete a slot, another public member function **`destroySlot`** is used.

We print the board after each player has played its turn using **`printBoard`** member function. Then, we continue with the other player for its turn. The game continues until we run out of rounds to play. When the game is over, the member function **`evaluateGame`** is called to determine the

game outcome by comparing the amount of x's and o's remained on the board. The player with the least number of pieces at the end of the game wins the game; of course a draw is also a possibility.

After the game is finished, you should delete the slots by calling the public member function **`clearBoard`** to avoid memory leaks.

The program flow detailed in this section has already been implemented in the main program and it is provided to you in the homework package. What you will do is to implement the **`Board`** class and its member functions.

The Data Structure to be Used

In this homework, you are asked to implement the doubly linked list data structure as a *class* (named **`Board`**) with four private data members, which are as follows; two **`slot`** pointers; one for ***`head`***, and another for ***`tail`*** node of the doubly linked list, two integers: **`xCnt`** and **`oCnt`** to keep track of the number of pieces of the players. The private part of the **`Board`** class will include the followings.

```
private:
    slot * head;
    slot * tail;
    int xCnt;
    int oCnt;
```

The **`slot`** struct of this data structure must have the following data members. It is sufficient to have a dummy constructor with no initializations.

```
struct slot
{
    slot * next;
    slot * prev;
    CharStack slotStack;

    slot ()
    {}
};
```

The **`next`** and **`prev`** pointers point to the next and previous slots respectively. **`slotStack`** is a stack of characters which is an object of a class called **`CharStack`**. This **`CharStack`** class is a static character stack class of which the constructor generates a stack with 4 elements capacity. **`CharStack`** class header and implementation have been provided to you.

An example abstract view of the data structure may be seen in Figure 3 with a given board configuration.

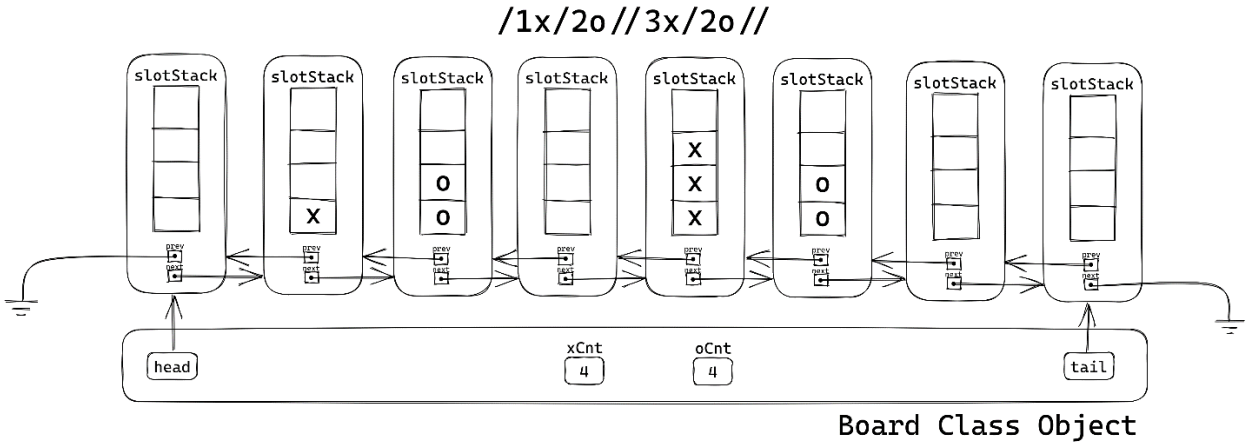


Figure 3: Doubly Linked List of Slot Data Structure

<u>No</u> other multiple data containers (built-in array, dynamic array, vector, matrix, 2D array, etc.) can be used. Do not use string data type to avoid these restrictions. The only place that you are allowed to use string is while printing the board (in the printBoard() function).

`Board` Class and Member Functions to be Implemented

The member functions that you will implement for the **`Board`** class are detailed below.

1. **Default constructor:** There will be a default constructor for the `Board` class that will not take any parameters. It will create an empty `Board` object with no slots in it by setting the `head` and `tail` pointers to null. Moreover, x and o counts will be set to zero.
2. **`noMove`:** This function takes two parameters, a character for the player and an integer for the steps to move. It returns true, if there is no possible valid move of that player; returns false otherwise (i.e. if there is at least one possible move). To do so, all slots and all move possibilities with the given number of steps are to be tried for that player. However, no movement is done in this function; its task is to check move possibility.
3. **`validMove`**: This function takes four parameters, (i) a character for the player, (ii) an integer for the starting index of the player's slot, (iii) an integer for the amount of steps to move, (iv) an integer for the direction of move (0 for left, 1 for right; no need for input check). The function returns a code (0, 1, 2, 3, or 4) as explained in the Program Flow section above. This function does not make the actual move.
   The indexing mechanism is not explicit for linked lists; thus, you will not keep any index value in the slot nodes. You will just count the slots starting zero, which is assumed to be the leftmost slot.
4. **`movePiece`**: This function takes the source and the target indices as parameters and completes the move operation via pop and push operations on the corresponding slot's stacks. It does not return anything. Indexing is explained in the previous function. Since we call this function after doing all validity controls, you do not need to make any extra checks.

5. **`printBoard`**: This function does not take any parameter and does not return anything. It displays the board as explained before. You may also check out the sample runs for the format of display.
   Hint for implementation: You have to display the content of the stacks line by line. So for each line, each slot's stack will contribute one character to the line. To accumulate these characters, you can utilize strings (but usage of strings is allowed for printing purposes only in this function; nowhere else in the code that you will write).
6. **`evaluateGame`**: This function does not take any parameters. It returns 1 if `x` has less pieces than `o`; returns 2 if `o` has less pieces than `x`; returns 3 if both have the same amount of pieces.
7. **`targetSlotFull`**: This function takes a slot index as parameter and returns true if the slot is full (i.e. slot's stack is full); returns false otherwise. Indexing logic is explained previously. You can assume that a valid index has been given as parameter.
8. **`destroySlot`**: This function takes a slot index as parameter and deletes that slot from the linked list. It also updates the piece counts accordingly. Here be careful about the linked list connections and special conditions of updating the head and tail of the Board class. Indexing logic is explained previously. You can assume that a valid index has been given as parameter.
9. **`createSlotBegin`**: This function takes two parameters. First one is the player character, say *ch*, and the second one is the number of characters, say *num*. The function does not return anything but creates a slot at the head of the linked list. This slot's stack contains *num* counts of *ch*. This function also updates the piece counts accordingly. Here, assume that *num* is 1, 2 or 3, and *ch* is either x or o; in other words, parameter values are not needed to be checked.
10. **`createSlotEnd`**: This function takes two parameters. First one is the player character, say *ch*, and the second one is the number of characters, say *num*. The function does not return anything but creates a slot at the tail of the linked list. This slot's stack contains *num* counts of *ch*. This function also updates the piece counts accordingly. Here, assume that *num* is 1, 2 or 3, and *ch* is either x or o; in other words, parameter values are not needed to be checked.
11. **`createEmptySlotEnd`**: This function does not take any parameter and does not return anything. It just adds an empty slot node at the end (tail) of the Board's linked list.
12. **`clearBoard`**: This function does not take any parameter and does not return anything. It deletes all of the slot nodes in the list.

Provided main cpp file, CharStack files and the requested class implementation

We have provided the main function of this homework to you within the homework pack. You have to use it directly and **without any modifications.** We also provided one header and one cpp file for the `CharStack` class. You will also use them **without any modifications**.

As you see from the provided main function, the construction of the data structure via calling class member functions and game logic are all implemented. What you have to do is to write the class' header and implementation files so that they work with the given main function. That means, you will write one header and one cpp file for the `Board` class.

The member functions used in main are self-explanatory and match with the game operations. We expect you explore the member function prototypes and requirements by analyzing the given main program and the explanations in this homework document.

If you need more member functions, other than the ones that we use in main, you are more than welcome to write. However, please first read the "Object Oriented Design Manifest" part below.

Object Oriented Design Manifest

We believe it is clear that you have to apply proper object oriented design principles in this homework, but our past experience says that most of the students are either unaware or not willing to follow these principles. Thus, we wanted to manifest some important rules about the class design and implementation.

1) According to the rule #1 of object oriented design, each member function **must be multi-purpose and must perform as single task as possible**. We picked the member functions used in main using this principle. You may add some extra member functions to be used internally for the implementations of the member functions (not in main since you are not allowed to change main), but please do not forget this principle while doing so.

2) Another important rule of object oriented programming is to hide the details of the class from the programmer that uses this class. Particularly for this homework, this means that in any free function and in main function, we should not reach the head and tail of the doubly linked list, and the piece counts directly. We have to make all of the updates, searches, etc. in main program through the member functions. Yes, technically it is possible and really easy to write a member function that returns head/tail so that we can freely manipulate the data structure in the main program, but this is totally against the spirit behind object oriented programming and **we do not do this**. Actually, since you are not allowed to change main, we guarantee that you will not manipulate the data structure using head/tail in main, but we wanted to make this clarification here.

3) There is also a hidden challenge for you to manipulate the stacks of the slots in Board class member functions. Since the stack is another class and you cannot reach the private data members of the stack class in Board member functions, the only way to manipulate (access, update, etc.) the stacks is via the member functions of the stack class. Thus, please check out the provided stack class for the provided member functions. And, please please please, do not search for workarounds to be able to reach/use the stack private data members in Board class (actually, there are such workarounds such as using *friend*, but we do not want this and if you do so, your homework will not be graded).

4) In a separate cpp file, you have to have the member function implementations. If you want to have some extra free functions to help the implementation of the member functions, also write them in this file (not in main file). If you will write some helper free functions, please remark that the rules #2 and #3 above are also valid for free functions.

5) Class and struct definitions will be in a header file.

6) You will **not** submit the provided main and stack files. You will submit only Board class' header and cpp (implementation) files. Please see the submission guidelines below.

Some Important Rules

In order to get a full credit, your programs must be efficient and well presented, presence of any redundant computation or unnecessary memory usage or bad indentation, or missing, irrelevant comments are going to decrease your grades. You also have to use understandable identifier names, informative introduction and prompts. Modularity is also important; you have to use functions wherever needed and appropriate. Since using classes is mandated in this homework, a proper object-oriented design and implementation will also be considered in grading.

Since you will use dynamic memory allocation in this homework, it is very crucial to properly manage the allocated area and return the deleted parts to the heap whenever appropriate. Inefficient use of memory may reduce your grade.

When we grade your homework, we pay attention to these issues. Moreover, in order to observe the real performance of your codes, we may run your programs in *Release* mode and **we may test your programs with very large test cases**. Of course, your program should work in *Debug* mode as well.

Please do not use any non-ASCII characters (Turkish or other) in your code (not even as comments). And also do not use non-ASCII characters in your file and folder names. We really mean it; otherwise, you may encounter some errors.

You are not allowed to use codes found somewhere online or other than course material. You can use only the material that are covered in lectures (CS201 and CS204 so far). However, if you use a non-standard C++ library/function/class (such as strutils) covered in the courses, either (i) you have to submit your code together with these extra source and header files so that CodeRunner runs your code properly, or (ii) copy necessary functions from them into your submitted code file by specifying where you copied them. In some assignments, we allow option (i) above in CodeRunner settings, but in some others we may not allow, so you can follow option (ii). In this homework, you may follow (ii).

You are allowed to use sample codes shared with the class by the instructor and TAs. However, you cannot start with an existing .cpp or .h file directly and update it; you have to start with an empty file. Only the necessary parts of the shared code files can be used and these parts must be clearly marked in your homework by putting comments like the following. Even if you take a piece of code and update it slightly, you have to put a similar marking (by adding "`and updated`" to the comments below.

```
/* Begin: code taken from ptrfunc.cpp */
…
/* End: code taken from ptrfunc.cpp */
```

Submission Rules (PLEASE READ SINCE SOME PARTS CHANGED)
(Some parts updated on April 1, 2023)

It'd be a good idea to write your name and last name in the files that you will submit (as a comment line of course). Do not use any Turkish characters anywhere in your code (not even in comment parts). For example, if your full name is "Satılmış Özbugsızkodyazaroğlu", then you must type it as follows:

// Satilmis Ozbugsizkodyazaroglu

We use CodeRunner in SUCourse for submission. No other way of submission is possible. Since the functionality part of the grading process will be automatic, you have to strictly follow these guidelines; otherwise, we cannot grade your homework.

You will see two different versions of the homework at SUCourse. One of them (**testing ground version**) is for you to test your code. However, you will **not** submit there. You will submit to the other version (**submission version**). You cannot test your code in the submission version. **We will grade what you submit to the submission version.** Please make sure that the submitted version is the final one that you tested and you submitted both `board.cpp` (by copying to the answer area) and `board.h` (as attachment on the user interface).

The advantage CodeRunner it is that you will be able to test your code against sample test cases. However, the output should be exact, but the textual differences between the correct output and yours can be highlighted (by pressing "show differences" button) on the interface.

The submission mechanism is a bit different than the other assignments. You will submit two files: `Board.cpp` and `Board.h` files. **`Board.cpp` file's content will be copied and pasted into the "Answer" area** as in other assignments. **However, you will upload `Board.h` file as attachment** to your submission in the relevant assignment submission page on SUCourse. Then you can test your code via CodeRunner against the sample runs (by pressing the "Check" button). You can check as much as you can; there is no penalty for multiple checks. However, as mentioned above, you can test only in the testing ground version of the homework.

The file name that you will upload must definitely be `Board.h`; otherwise it does not work. And you will not upload the provided main and stack files; we have already put them there.

Even any tiny change in the output format will result in your grade being zero (0) for that particular test case, so please test your programs yourself, and against the sample runs that are available at the relevant assignment submission page on SUCourse (CodeRunner).

In the CodeRunner, there are some visible test cases. However, they may not be used in grading. The grading test cases might be different and it is always a possibility that your program works with these given test cases, but does not work with one or more grading test cases. Thus you have to test your program thoroughly.

You don't have to complete your task in one time, you can continue from where you left last time, but you should not proceed with submission before finalizing it. Therefore, you should make sure that it's your final solution version before you submit it. Also, we still do not suggest that you develop your solution on CodeRunner but rather on your IDE on your computer.

So far the operations that you can perform on the **testing ground version** of the homework has been explained. Once you decide to submit, you should switch to **submission version** of the homework and complete the submission process there (do not submit to the testing ground version). In the submission version, you will not be able to test your homework. In the submission version, you should just copy the content of the `board.cpp` into the answer area, and upload `board.h` file to the attachments part, and then complete the submission process by finishing the attempt. Of course, you have to use the final version of your code during submission. Moreover, remark that you have to manually "Submit" (there is no automatic submission). There is no re-submission. That means, after you submit, you cannot take it back.

Last, even if you cannot completely finish your homework, you can still submit.

Sample Runs

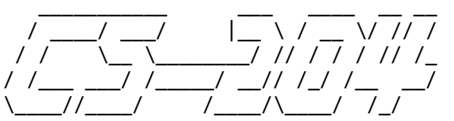
Sample runs are given below, but these are not comprehensive, therefore you have to consider **all possible cases** to get full mark. Inputs are shown in ***bold and italic***.

We will configure CodeRunner to test these sample runs for you. However, grading test cases might be totally different.

Test Case 0 (Deleting at the head)

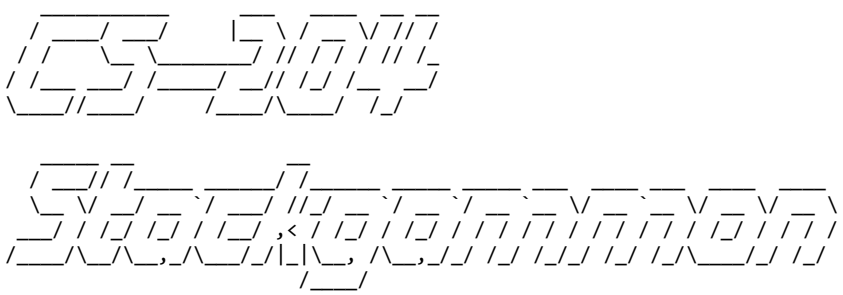
```
  ___________      ___   ____  __ __
  / ____/ ___/     |__ \ / __ \/ // /
 / /    \__ \_______/ // / / / // /_
/ /___ ___/ /_____/ __// /_/ /__  __/
\____//____/     /____/\____/  /_/

  _____ __                __
  / ___// /_____ _____/ /_____ _____ ____ ___  ____ ___  ____  ____
  \__ \/ __/ __ `/ ___/ //_/ __ `/ __ `/ __ `__ \/ __ `__ \/ __ \/ __ \
 ___/ / /_/ /_/ / /__/ ,< / /_/ / /_/ / / / / / / / / / / /_/ / / / /
/____/\__/\__,_/\___/_/|_|\__, /\__,_/_/ /_/ /_/_/ /_/ /_/\____/_/ /_/
                         /____/
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
3x//2x/2o//3o

x    o
x xo o
x xo o
^^^^^^
[*] Enter a round limit:
1
[*] Enter a random number generator seed:
1
[*] Round 0
[x] X's turn:
Throwing die...
Die is 2.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
2 0

    o
  o o
 xo o
^^^^^
[o] O's turn:
Throwing die...
Die is 4.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
4 0


  o o
oxo o
^^^^^
[*] Game over!
  _   __    _       ___                 __
  | |/ /   | |     / (_)___  ____/ /
  |   /    | | /| / / / __ \/ ___/ /
 /   |     | |/ |/ / / / / (__  )_/
/_/|_|     |__/|__/_/_/ /_/____(_)
```

Test Case 1 (Deleting at the tail)



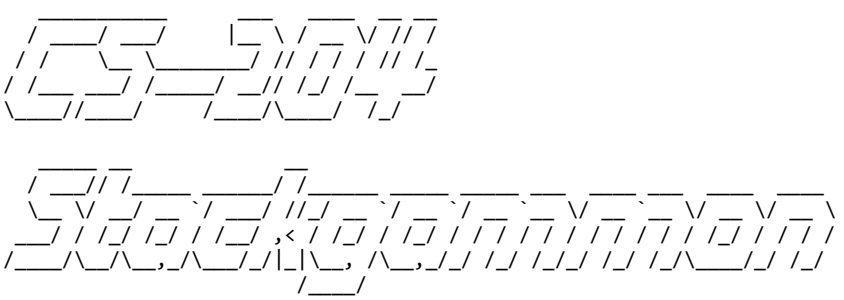
```
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
3x//2x/2o//3o

x    o
x xo o
x xo o
^^^^^^
[*] Enter a round limit:
1
[*] Enter a random number generator seed:
2
[*] Round 0
[x] X's turn:
Throwing die...
Die is 4.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1

     o
x xo o
x xoxo
^^^^^^
[o] O's turn:
Throwing die...
Die is 2.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
3 1


x x
x xox
^^^^^
[*] Game over!
```

Test Case 2 (Deleting at the middle)



```
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
1x/2x/3x/3o/2o/1o

  xo
 xxoo
xxxooo
^^^^^^
[*] Enter a round limit:
1
[*] Enter a random number generator seed:
1
[*] Round 0
[x] X's turn:
Throwing die...
Die is 2.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1

  o
 xoo
 xooo
^^^^^
[o] O's turn:
Throwing die...
Die is 4.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
4 0

  o
 xoo
oxoo
^^^^^
[*] Game over!
```

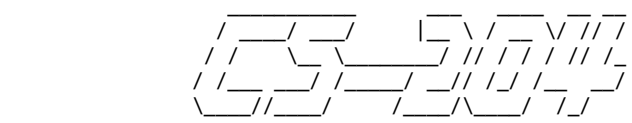
Test Case 3 (Deleting at the tail and appending to the head)

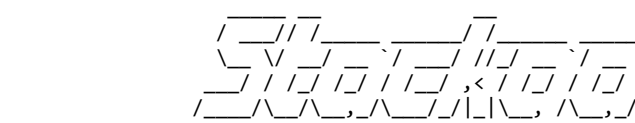


```
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
1x/3o/3x/3o/3o/3o

 oxooo
 oxooo
xoxooo
^^^^^^
[*] Enter a round limit:
2
[*] Enter a random number generator seed:
1
[*] Round 0
[x] X's turn:
Throwing die...
Die is 2.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1

 oooo
 oooo
 oooo
^^^^^
[o] O's turn:
Throwing die...
Die is 4.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
4 0

 ooo
 oooo
ooooo
^^^^^
[*] Round 1
[x] X's turn:
Throwing die...
Die is 2.
[x] You have no legal moves! Choose to append at start (1) or at the end (0):
1

  ooo
  oooo
xooooo
^^^^^^
[o] O's turn:
Throwing die...
Die is 4.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
1 1

  oooo
  oooo
x oooo
^^^^^^
```

```
[*] Game over!
```

Test Case 4 (Delete at the tail and appending at the tail)





```
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
2x/1o/2x/3x/3x/3x/3o

   xxxo
x xxxxo
xoxxxxo
^^^^^^^
[*] Enter a round limit:
2
[*] Enter a random number generator seed:
16
[*] Round 0
[x] X's turn:
Throwing die...
Die is 3.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1

   xxo
  xxxo
xoxxxo
^^^^^^
[o] O's turn:
Throwing die...
Die is 4.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
1 1

   xx
  xxx
x xxx
^^^^^
[*] Round 1
[x] X's turn:
Throwing die...
Die is 3.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1

   x
  xx
  xx
^^^^
[o] O's turn:
```

```
Throwing die...
Die is 4.
[o] You have no legal moves! Choose to append at start (1) or at the end (0):
0

   x
  xx
  xxo
^^^^^
[*] Game over!
  ____    _      ___           __
  / __ \   | |     / (_)___  _____/ /
 / / / /   | | /| / / / __ \/ ___/ /
/ /_/ /    | |/ |/ / / / / (__  )_/
\____/     |__/|__/_/_/ /_/____(_)
```

Test Case 5 (Showing error messages and appending at the front)

```
  ___________     ___   ____  __ __
  / ____/ ___/     |__ \ / __ \/ // /
 / /    \__ \_______/ // / / / // /_
/ /___ ___/ /_____/ __// /_/ /__  __/
\____//____/     /____/\____/  /_/

  _____ __              __
  / ___// /_____ _____/ /_____ _____ ___  ____ ___  ____ ____
  \__ \/ __/ __ `/ ___/ //_/ __ `/ __ `/ __ `__ \/ __ `__ \/ __ \/ __ \
 ___/ / /_/ /_/ / /__/ ,< / /_/ / /_/ / / / / / / / / / / /_/ / / / /
/____/\__/\__,_/\___/_/|_|\__, /\__,_/_/ /_/ /_/_/ /_/ /_/\____/_/ /_/
                         /____/
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
/2x/3x/2o/3o/1o

  x o
 xxoo
 xxooo
^^^^^^
[*] Enter a round limit:
2
[*] Enter a random number generator seed:
204
[*] Round 0
[x] X's turn:
Throwing die...
Die is 4.
[x] You have no legal moves! Choose to append at start (1) or at the end (0):
1

   x o
  xxoo
x xxooo
^^^^^^^
[o] O's turn:
Throwing die...
Die is 3.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
6 0
[!] Target index not available. Try again...
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
5 0
[!] Target index not available. Try again...
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
7 0
[!] Choice index out of bounds. Try again...
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
```

```
4 0

   x o
  xx o
xoxxooo
^^^^^^^
[*] Round 1
[x] X's turn:
Throwing die...
Die is 4.
[x] You have no legal moves! Choose to append at start (1) or at the end (0):
0

   x o
  xx o
xoxxooox
^^^^^^^^
[o] O's turn:
Throwing die...
Die is 3.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
1 1

   x o
  xxoo
x xxooox
^^^^^^^^
[*] Game over!
  ____    _      ___           __
 / __ \   | |     / (_)___  _____/ /
/ / / /   | | /| / / / __ \/ ___/ /
/ /_/ /    | |/ |/ / / / / (__  )_/
\____/     |__/|__/_/_/ /_/____(_)
```

Test Case 6 (validMove() checks)

```
  ___________     ___   ____ __ __
 / ____/ ___/     |__ \ / __ \/ // /
/ /    \__ \_______/ // / / / // /_
/ /___ ___/ /_____/ __// /_/ /__  __/
\____//____/     /____/\____/  /_/

  _____ __              __
 / ___// /_____ ______/ /_____ _____ _____ ___  ____ ___  ____  ____
 \__ \/ __/ __ `/ ___/ //_/ __ `/ __ `/ __ `__ \/ __ `__ \/ __ \/ __ \
___/ / /_/ /_/ / /__/ ,< / /_/ / /_/ / / / / / / / / / / /_/ / / / /
/____/\__/\__,_/\___/_/|_|\__, /\__,_/_/ /_/ /_/_/ /_/ /_/\____/_/ /_/
                         /____/
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
/1x/1o//1o/1x/



 xo ox
^^^^^^^
[*] Enter a round limit:
2
[*] Enter a random number generator seed:
204
[*] Round 0
[x] X's turn:
Throwing die...
Die is 4.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
2 1
[!] Choice index not yours to move! Try again...
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
```

```
3 1
[!] Choice index not yours to move! Try again...
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
1 0
[!] Target index out of bounds. Try again...
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
4 1
[!] Choice index not yours to move! Try again...
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
5 1
[!] Target index out of bounds. Try again...
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
5 0


 x
 xo o
^^^^^^^
[o] O's turn:
Throwing die...
Die is 3.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
2 1


 x
 x  oo
^^^^^^^
[*] Round 1
[x] X's turn:
Throwing die...
Die is 4.
[x] You have no legal moves! Choose to append at start (1) or at the end (0):
0


 x
 x  oo x
^^^^^^^^
[o] O's turn:
Throwing die...
Die is 3.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
4 0
[!] Target index not available. Try again...
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
5 0


 x
 xo o  x
^^^^^^^^
[*] Game over!
   ____     _       ___                __
  / __ \   | |     / (_)___  _____/ /
 / / / /   | | /| / / / __ \/ ___/ /
/ /_/ /    | |/ |/ / / / / (__  )_/
\____/     |__/|__/_/_/ /_/____(_)
```

Test Case 7 (Large Seed, large round number, tie result)

```
  _____________      ___   ____  __ __
  / ____/ ___/     |__ \ / __ \/ // /
 / /    \__ \_______/ // / / / // /_
/ /___ ___/ /_____/ __// /_/ /__  __/
\____//____/     /____/\____/  /_/

   _____ __                 __
  / ___// /_____ _____/ /_____ _____ _____ ___  ____ ___  ____  ____
  \__ \/ __/ __ `/ ___/ //_/ __ `/ __ `/ __ `__ \/ __ `__ \/ __ \/ __ \
 ___/ / /_/ /_/ / /__/ ,< / /_/ / /_/ / / / / / / / / / / /_/ / / / /
/____/\__/\__,_/\___/_/|_|\__, /\__,_/_/ /_/ /_/_/ /_/ /_/\____/_/ /_/
                         /____/
[*] Welcome to the Stackgammon Game!
[*] Enter a board configuration:
3x/3x//1x/2x/1x/3x/3x/3o/3o/3o/3o/1o/2o//1o

xx    xxoooo
xx  x xxoooo o
xx xxxxxooooo o
^^^^^^^^^^^^^^^
[*] Enter a round limit:
7
[*] Enter a random number generator seed:
16384
[*] Round 0
[x] X's turn:
Throwing die...
Die is 4.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
3 1

xx    xoooo
xx  x xoooo o
xx  xxxooooo o
^^^^^^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 1.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
7 1

xx    x oo
xx  x xooo o
xx  xxxoooo o
^^^^^^^^^^^^^
[*] Round 1
[x] X's turn:
Throwing die...
Die is 1.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
5 1

xx     oo
xx  x ooo o
xx  x ooooo o
^^^^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 3.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
```

```
10 0

xx     o
xx  x oo
xx  x oooo o
^^^^^^^^^^^
[*] Round 2
[x] X's turn:
Throwing die...
Die is 1.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1

      o
x  x oo
x  x oooo o
^^^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 1.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
7 0


x  x o
x  x o o o
^^^^^^^^^^
[*] Round 3
[x] X's turn:
Throwing die...
Die is 3.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
3 0

x
x    o
x  x o o o
^^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 4.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
9 0

x    o
x    o
x  x o o
^^^^^^^^^^
[*] Round 4
[x] X's turn:
Throwing die...
Die is 3.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
3 0

    o
    o
    o o
^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 2.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
6 0



^^^^^^^^
[*] Round 5
```

```
[x] X's turn:
Throwing die...
Die is 4.
[x] You have no legal moves! Choose to append at start (1) or at the end (0):
1


x
^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 1.
[o] You have no legal moves! Choose to append at start (1) or at the end (0):
0


x        o
^^^^^^^^^^
[*] Round 6
[x] X's turn:
Throwing die...
Die is 4.
[x] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
0 1


    x    o
^^^^^^^^^^
[o] O's turn:
Throwing die...
Die is 2.
[o] Make your move. State the index (starts from 0) of your piece you want to move, and right (1) or left (0):
9 0


    x  o
^^^^^^^^^^
[*] Game over!
  _______     __
 /_  __(_)__  / /
  / / / / _ \/ /
 / / / /  __/_/
/_/ /_/\___(_)
```

Good Luck!
Albert Levi, Selim Kırbıyık