Sabanci University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Summer 2022-2023

Take-Home Exam 2
Space Colony Management using Doubly Linked List

Due: 14 August 2023 11.55pm (SHARP)

DISCLAIMER:

Only checking the sample run cases might not be sufficient as your solution will be checked against a variety of samples different from the provided samples; however checking and studying these cases is highly encouraged and recommended.

You can NOT collaborate with your friends and discuss your solutions with each other. You have to write down the code on your own. Plagiarism will not be tolerated AND cooperation is not an excuse!

Friendly note:

Take Home Examinations aim at practicing the topics you've seen during the course so that you have a better understanding of what you've learnt. Using any concept that you haven't covered in either CS201 or in CS204 yet, is restricted to avoid backdoors for the challenges that you should overcome with what you've seen already. If you proceed without considering this limitation, you will most probably be asked to have an oral interview, and the result may affect your THE grade.

Aim

The aim of this take home exam (THE) is to practice the topic of linked lists, specifically Doubly Linked Lists (DLL). You are highly encouraged to study these topics well before starting the task to save time and avoid much frustration.

Introduction

This idea of this THE is an adaptation of THE1. In this THE, you will implement the basic tasks of the colony management system using a DLL. You are still going to read data from 3 different files about stock, consumption, and an initial setup of the colony. For the sake of simplicity, one major difference in this THE from the previous one is that the colony structure will be only one-dimensional.

Note on Implementation:

You ***must*** use a DLL as the main container to store and manipulate the data; otherwise, your grade will be automatically converted to zero (0).
Below, we are providing the basic structure of three structs that you need to use while implementing the three DLLs to store the data from the 3 input files.

Note: feel free to enrich them more, if you see a need for that.

```
struct stockNode{
  string resourceName;
  int resourceQuantity;
  stockNode *next;
  stockNode *prev;
};
```

```
struct consumpNode{
  char buildType;
  vector<int> consumpQtys;
  consumpNode *next;
  consumpNode *prev;
};
```

```
struct colonyNode{
  char buildType;
  int emptyBlocks2TheLeft;
  colonyNode *next;
  colonyNode *prev;
};
```

For the `colonyNode`, as you will see while checking the colony input files, every colony file contains one line as a combination of (lower- or upper-case) English characters in addition to dashes '-'. Dashes represent the empty blocks within the colony line.
Thus, the variable `emptyBlocks2TheLeft` is going to be used to store the number of empty blocks (dashes) to the left of each building. For some concrete examples, please check the sample files and sample runs.
An important note here is that we assume that the line can be extended from the tail side as much as we can, which means that we can keep adding nodes to the end of the colony line.

Input files and User Inputs

As was the case with THE1, your program will read inputs from 3 different text files, usually having the following naming conventions:

1. *stockX.txt*
   - e.g., stock1.txt, stock2.txt, …
   - This type of file includes available resources in the following format

```
resource1_name resource1_quantity
resource2_name resource2_quantity
.
.
resourceN_name resourceN_quantity
```

   - In each line of this file, every resource_name is a one-word string, while every resource_quantitiy is a non-negative integer number.
   - In each line of this file, every pair of the resource_name and resource_quantitiy are separated with a single whitespace.
   - There won't be any duplicate resource names or duplicate pairs of resources_names and resource quantities. i.e., every resource name (with its corresponding quantity) will appear only once in a single file.

2. *consumptionX.txt*
   - e.g., consumption1.txt, consumption2.txt, …
   - This type of text file has the types of buildings that can be built in the current colony, along with lists of *N* non-negative integers that indicate the amount of each of the *N* resources required to build a block (one cell on the colony) of that type of building.

```
buildingType1 required_quantity_resource1 … required_quantity_resourceN
buildingType2 required_quantity_resource1 … required_quantity_resourceN
.
.
```

   - Each buildingType is a single uppercase or lowercase English letter.
     - Note: building *'A'* is different from building *'a'*. So, the input from this file should be treated in a case-sensitive manner.
   - For each buildingType, its corresponding list of *N* non-negative integers indicates the amount of each of the *N* resources required to build a block of that given building type (*following the same order of resource names given in the first text file that contains the available resources information (i.e., stockX.txt)*).

- There won't be any duplicate building types or duplicate pairs of building types and required resources quantities. i.e., every building type (with its corresponding required resources quantities) will appear only once in a single file.

3. *colonyX.txt*
   - e.g., `colony0.txt`, `colony1.txt`, …
   - This type of text file will have ***only one line*** representing the buildings that will occupy some blocks (cells) of the colony ***line*** when the colony is to be loaded.
   - Each cell in the colony line is going to be either '-', which indicates that this specific cell is empty, or any other uppercase or lowercase English letter, which indicates that this cell is occupied by a block of that building type.
     - Each line will end with an English letter. i.e., there will not be any dashes '-' at the end of line.
     - All the uppercase or lowercase letters that are found in the colony file are also mentioned in the second file (*consumptionX.txt*), where their resources consumption was mentioned. So, you don't need to check for such irregularities.

None of the files will have extra whitespaces or empty lines. That is, you don't have to check for such irregularities.

Notes on files naming conventions:

In general, file names can be any valid names that the user can use to name a file. For example, a text file that contains the stock information can be named as "`myStock.txt`", "`st.txt`" or any other valid name. Same applies on the other types of files that you are going to deal with in this THE. However, you don't have to worry about that for this THE as you are not expected to check for file naming patterns to verify that you are reading the data from the correct file.

Also, the program will read data from three files, and the content of each file is different from the others. Reading one file's contents in place of another one might lead to unintended behavior. However, again, you don't have to worry about that for this THE as we are not going to test such a case. That is, there won't be a case where we provide a name for a file that contains consumption data when the program is expecting to receive the file name for the colony or the stock data, for example.

The case that your program should check for file opening failure is when there is a problem with opening the file due to an incorrect file name given by the user.

Note on User Inputs Sase-Sensitivity:

When dealing with the data from the files, and later in the program, or when the user enters any input, you should be operating on the case-sensitive mode. That is, 'A' as a building type is different from 'a'. Same applies for other similar cases.
Your program should treat all the input characters (letters) and strings in a case-sensitive manner. That is, an uppercase character 'D', is not the same as lowercase 'd'. Please refer to sample runs for more examples.

Program Flow

You are encouraged to carefully check the sample runs as they complement the textual description, demonstrate different cases that your code should handle, and they also show the expected behavior of your program.

Your program should start by asking the user for the stock data file name, then ask for the consumption data file name, and finally asking for the colony file name.
Of course, whenever your program receives a file name, it should try to open it and read its contents and store it in the respective DLL. As usual, if your program didn't manage to open any of the files, your program should keep asking for a valid name until the user enters it.

The process for the first two files are almost the same. Your program basically should:

- start by asking the user to enter the file name (keep asking if the file name is not correct)
- open this file, read and store its content in a DLL,

The above steps need to be followed for reading the first two files' contents (i.e., `stockX.txt`, and `consumptionX.txt`).

Then, we come to the 3rd file (`colonyX.txt`), which contains the colony's initial setup. When reading the content of this file, and trying to construct the buildings, your program will be using (consuming) resources from the resources stock, and therefore, you are limited to the initial resource quantities that you read from the resource stock (`stockX.txt`) in the very beginning. Thus, every building to be constructed will consume a portion of the resources that it requires to be built

(consumption quantities of resources for each building type are given in the *consumptionX.txt* text file).

If any of the resources was not sufficient for constructing any building in the given initial colony setup, your program should print specific error message, clear the allocated memory and exit without further executing the rest of the code (sample run 7 demonstrates such a case)

If, on the other hand, your program manages to load the colony (i.e. there are enough resources for all the buildings in the colony), then your program should proceed with displaying the menu interface with 8 different options which are shown in a subsequent section in this document .

An important condition regarding storing the colony line

You can just store the buildings as nodes in the colony DLL. In other words, **you don't necessarily need to (and hence encouraged not to) store the empty blocks (dashes) in the DLL nodes.**

Menu interface

Your program will display to the user the menu with 8 options below, and then ask the user to enter an option number and based on that input, your code should execute the corresponding functionality. Below are the 8 options with a description for each of them.

1. *Construct a new building on the colony*

   When the user wants to construct a new building, he/she needs to enter two inputs:
   1. The type of the building
      a. Here, you need to validate that the building type the user entered is valid. A valid building type is one that is available in the consumption DLL. So you will search for that building type in the consumption DLL (given that you stored consumption file contents in this DLL), and if you found it, you will proceed with asking for the second input (the position). If, on the other hand, the building type was not valid, you will keep asking the user to enter a valid one.
   2. The order (index) of the empty block on which they want to build it.
      a. Note that the empty blocks are indexed starting with 1 from left (starting with the first empty block) and going to the right.

b. As you noted, we have the colony represented as one line, so a valid index would be a positive integer.
   i. *1* represents the first empty block; *2* represents the second empty block, etc.
   ii. There is no upper limit for the index that the user can enter, which means even if the colony has only (let say) *7* actual buildings constructed, and the user entered an index of a new building to be built as (let say) *15,* your program should accept this as a valid index, and build that building and account for all those empty blocks between the last actual block in the colony and the newly created one. You will find some more examples in the sample runs. Sample run 9 demonstrates a similar case.
   iii. We assume that the user will always enter a valid index; so you don't need to check for that.

After successfully adding the building your program should print a message. Please refer to sample runs for specific messages

If there are not enough resources for constructing the building, the program should print a couple of messages and go back to the menu (to get another menu option input). Sample run 9 demonstrates this case.

In the figure below we show an example of an initial colony setup and the effect of adding new buildings on the DLL structure. This figure is a demonstration of sample run *4*.

Initially the *colony0.txt* file has the following line:
`--HSS`

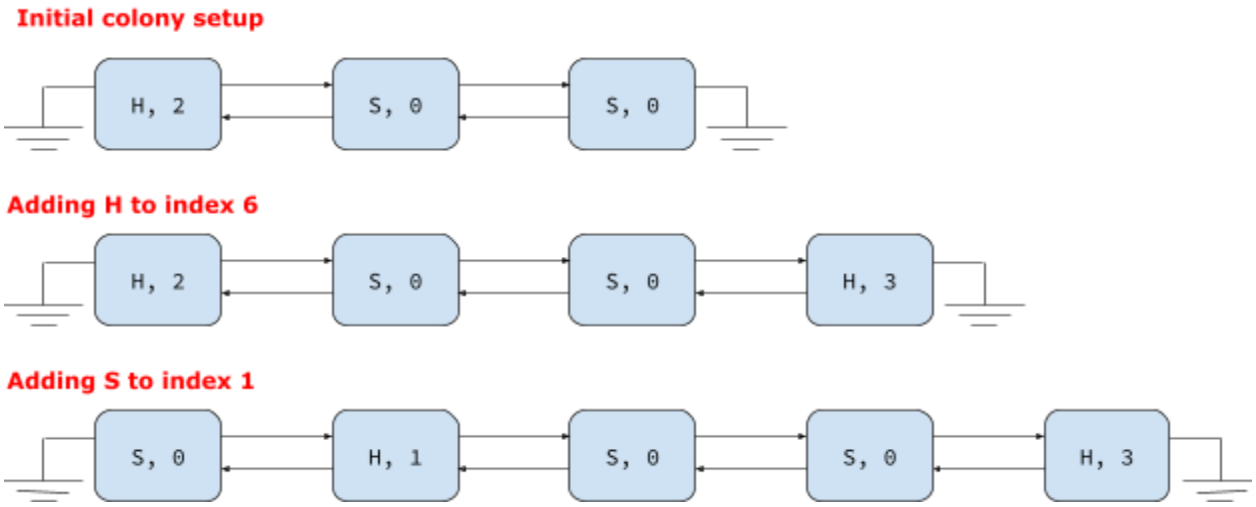
That colony is going to be stored in the colony DLL as shown in the first version of the DLL in the figure. Note that the node at the head (which contains `H, 2`) is encoding (i.e, representing) the two dashes that came before it as number 2 (recall the variable `emptyBlocks2TheLeft` in initial `colonyNode` struct that we provided in the beginning of this document) .

Then, after adding building of type H at index 6, the colony now can be shown as follows:
`--HSS---H`
Note that the basic information of the first three nodes stayed as they were, and the newly created node with building type H and 6 as the index (order) of the

empty block has been added to the end of the DLL (from tail side) with 3 as the number of empty blocks on its left. Note that in this step, you will have to update the tail of the DLL, as well as the `next` pointer of the last node as you usually do when adding the to tail of a DLL.

Then, after adding building of type S at index 1, the colony now can be shown as follows:
`S-HSS---H`

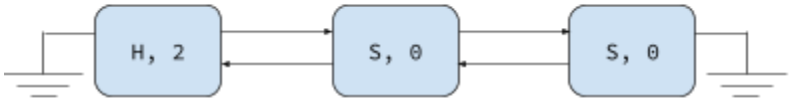
Here, please note the change in the number of empty blocks to the left of the first node with H. Also, you need to do the needed arrangements that you usually do when adding a node to the beginning (head) of a DLL.



2. *Destruct/Disassemble a building from the colony*
    - Your program should ask for the type of the building that needs to be destroyed.
        - If there is no building on the colony of the type that the user has entered, your program should output a specific message and go back to the menu (to get another menu option input).
        - If, on the other hand, your program can find an instance of this building type on the colony, it should delete *only the first occurance* of the building (going from the left (head) of the DLL).
    - After deleting the building, your program should:
        - Print a specific message
        - Put back the resources that were used to build that building to the stock. (check sample run 9 for this case)

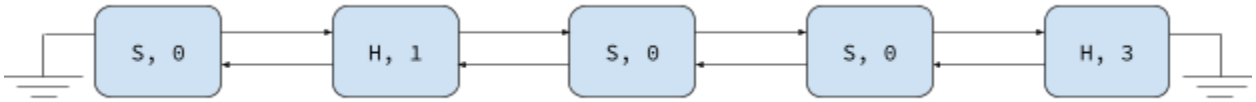
The figure below is an extension of the first figure to show the deletion of the first occurance of building *H*. Please note the change in the structure of the DLL, and in the number of empty blocks to the left of the second *S* node. When the building with type *H* was deleted, its position will be considered as an empty block. That is the reason the number of empty blocks to the left of the second *S* node has changed from zero to 2. In detail, the *H* block had an empty block to its left, and when it was deleted, it left its block as an empty one as well. Thus, now (after deleting *H*) The second node (that contains *S*) has 2 empty blocks to its left.
Please check sample run 5 as it demonstrates this case.

Initial colony setup



Adding H to index 6

Adding S to index 1



Destructing Building H

3. *Print the colony*

- When this option is selected, your program should print the colony line in two different ways:
  i. Print the colony line starting from the head of the colony DLL to the tail of it (showing only the buildings)
  ii. Print the colony line starting from the head of the colony DLL to the tail of it with the number of empty blocks (dashes) that precedes each block represented as a number in parenthesis to the left of the building.

4. *Print the colony in reverse*

Your program should print the colony line in the reverse order (starting from right going to left, i.e., starting from the DLL tail to its head).

5. *Print the colony while showing inner empty blocks*

   Print the colony (from left to right) while showing the inner empty blocks as dashes.

6. *Print the colony while showing inner empty blocks in reverse*

   Print the colony in the reverse order (from right to left) while showing the inner empty blocks as dashes.

7. *Print the stock*

   Print each resource followed by the available quantity between parentheses. The order of the output should follow the same order of the stockX.txt file, where each resource with its (available quantity) is printed on a separate line.

8. *Exit*

   Your program should clear the dynamic memory that was allocated throughout the program, print a message and then terminate the program.

Memory Management

As you will be allocating memory from the heap (free store) you are responsible for wisely managing the memory that your program allocates. That is one of the reasons that we asked you not to store the empty blocks information (dashes) while reading the colonyX.txt files. What you will be doing by just storing the actual buildings and representing the empty blocks information as just integer numbers is similar to one of the good practices that are being used in data compression and is sometimes used in storing sparse structures.
Moreover, after you are done with the user's requests, you need to free the memory that your program allocated for the DLLs. As we learnt, that is the good practice of freeing the dynamically allocated memory (whether through destructors, if you were implementing the DLL as class, or through a normal clearDLL() functions as you learned in the class and recitations)
Lack of proper memory management in your program may lead to loss of points in your final grade for this THE.

Note on Code Modularity

You should always write modular code. Among other benefits of this practice, it allows reusing certain code blocks which will make your code shorter and saves you from re-coding similar functionalities over and over again.

Sample Runs

Below, we provide some sample runs of the program that you will develop. The **bold** phrases are the standard input (cin) taken from the user (i.e., like **this**). You have to display the required information in the same order and with the same words/spaces as here; in other words, there must be an exact match!

You will submit your code through CodeRunner, and you can always use the "Show differences" feature to make it easier for you to figure out the mismatches in your output, if any.

Sample Run 1:

Please enter the stock file name:
stock5.txt
Please enter the consumption file name:
consumption5.txt
Please enter the colony file name:
colony5.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
7
Stock DLL:
Iron(47977)
Copper(4939)
Silicon(6811)
3
Colony DLL:
HCSEnPRL

(1)H(0)C(0)S(0)E(3)n(0)P(2)R(1)L
8
Clearing the memory and terminating the program.

Sample Run 2:

Please enter the stock file name:
stock5.txt
Please enter the consumption file name:
consumpt5.txt
Unable to open the file consumpt5.txt. Please enter the correct consumption file name:
consumption5.txt
Please enter the colony file name:
colooony5.txt
Unable to open the file colooony5.txt. Please enter the correct colony file name:
colony5.tx
Unable to open the file colony5.tx. Please enter the correct colony file name:
colony5.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
5
Colony DLL:
-HCSE---nP--R-L
8
Clearing the memory and terminating the program.

Sample Run 3:

Please enter the stock file name:
stock5.txt
Please enter the consumption file name:
consumption5.txt
Please enter the colony file name:
colony5.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony

3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
1
Please enter the building type:
X
Building type X is not found in the consumption DLL. Please enter a valid building type:
x
Building type x is not found in the consumption DLL. Please enter a valid building type:
H
Please enter the index of the empty block where you want to construct a building of type H
1
Building of type H has been added at the empty block number: 1
3
Colony DLL:
HHCSEnPRL
(0)H(0)H(0)C(0)S(0)E(3)n(0)P(2)R(1)L
8
Clearing the memory and terminating the program.

Sample Run 4:

Please enter the stock file name:
stock5.txt
Please enter the consumption file name:
consumption5.txt
Please enter the colony file name:
colony0.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
3
Colony DLL:
HSS
(2)H(0)S(0)S

5
Colony DLL:
--HSS
1
Please enter the building type:
H
Please enter the index of the empty block where you want to construct a building of type H
6
Building of type H has been added at the empty block number: 6
3
Colony DLL:
HSSH
(2)H(0)S(0)S(3)H
5
Colony DLL:
--HSS---H
1
Please enter the building type:
S
Please enter the index of the empty block where you want to construct a building of type S
1
Building of type S has been added at the empty block number: 1
3
Colony DLL:
SHSSH
(0)S(1)H(0)S(0)S(3)H
5
Colony DLL:
S-HSS---H
8
Clearing the memory and terminating the program.

Sample Run 5:

Please enter the stock file name:
stock5.txt
Please enter the consumption file name:
consumption5.txt
Please enter the colony file name:
colony0.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony

4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
3
Colony DLL:
HSS
(2)H(0)S(0)S
5
Colony DLL:
--HSS
1
Please enter the building type:
H
Please enter the index of the empty block where you want to construct a building of type H
6
Building of type H has been added at the empty block number: 6
3
Colony DLL:
HSSH
(2)H(0)S(0)S(3)H
5
Colony DLL:
--HSS---H
1
Please enter the building type:
S
Please enter the index of the empty block where you want to construct a building of type S
1
Building of type S has been added at the empty block number: 1
3
Colony DLL:
SHSSH
(0)S(1)H(0)S(0)S(3)H
5
Colony DLL:
S-HSS---H
2
Please enter the building type:
H
The building of type H has been deleted from the colony.
3
Colony DLL:

SSSH
(0)S(2)S(0)S(3)H
5
Colony DLL:
S--SS---H
8
Clearing the memory and terminating the program.

Sample Run 6:

Please enter the stock file name:
stock1.txt
Please enter the consumption file name:
consumption1.txt
Please enter the colony file name:
colony1.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
7
Stock DLL:
O2(700)
H2O(410)
Food(625)
Iron(264)
4
(Reverse) Colony DLL:
MKVGeZ
8
Clearing the memory and terminating the program.

Sample Run 7

Please enter the stock file name:
stock4.txt
Please enter the consumption file name:
consumption4.txt

Please enter the colony file name:
colony4.txt
Insufficient resource Titanite
Failed to load the colony due to insufficient resources.
Clearing the memory and terminating the program.

Sample Run 8

Please enter the stock file name:
stock3.txt
Please enter the consumption file name:
consumption3.txt
Please enter the colony file name:
colony3.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
3
Colony DLL:
BbCRM
(13)B(0)b(0)C(0)R(1)M
4
(Reverse) Colony DLL:
MRCbB
5
Colony DLL:
----------BbCR-M
6
(Reverse) Colony DLL:
M-RCbB----------
7
Stock DLL:
Biofuel(349)
Self-ReplicatingBots(21)
SolarPanels(238)
Hydrogel(51)
Nanomaterials(65)
8
Clearing the memory and terminating the program.

Sample Run 9

Please enter the stock file name:
stock3.txt
Please enter the consumption file name:
consumption3.txt
Please enter the colony file name:
colony3.txt
Please enter your choice:
1. Construct a new building on the colony
2. Destruct/Disassemble a building from the colony
3. Print the colony
4. Print the colony in reverse
5. Print the colony while showing inner empty blocks
6. Print the colony while showing inner empty blocks in reverse
7. Print the stock
8. Exit
1
Please enter the building type:
b
Please enter the index of the empty block where you want to construct a building of type b
1
Building of type b has been added at the empty block number: 1
3
Colony DLL:
bBbCRM
(0)b(12)B(0)b(0)C(0)R(1)M
5
Colony DLL:
b----------BbCR-M
1
Please enter the building type:
c
Building type c is not found in the consumption DLL. Please enter a valid building type:
C
Please enter the index of the empty block where you want to construct a building of type C
1
Building of type C has been added at the empty block number: 1
3
Colony DLL:
bCBbCRM
(0)b(0)C(11)B(0)b(0)C(0)R(1)M
5

Colony DLL:
bC----------BbCR-M
7
Stock DLL:
Biofuel(348)
Self-ReplicatingBots(20)
SolarPanels(237)
Hydrogel(50)
Nanomaterials(64)
1
Please enter the building type:
A
Insufficient resource Biofuel
Failed to add the building due to insufficient resources.
7
Stock DLL:
Biofuel(348)
Self-ReplicatingBots(20)
SolarPanels(237)
Hydrogel(50)
Nanomaterials(64)
1
Please enter the building type:
m
Building type m is not found in the consumption DLL. Please enter a valid building type:
M
Insufficient resource SolarPanels
Failed to add the building due to insufficient resources.
7
Stock DLL:
Biofuel(348)
Self-ReplicatingBots(20)
SolarPanels(237)
Hydrogel(50)
Nanomaterials(64)
1
Please enter the building type:
B
Please enter the index of the empty block where you want to construct a building of type B
15
Building of type B has been added at the empty block number: 15
7
Stock DLL:
Biofuel(298)

Self-ReplicatingBots(14)
SolarPanels(221)
Hydrogel(21)
Nanomaterials(30)
3
Colony DLL:
bCBbCRMB
(0)b(0)C(11)B(0)b(0)C(0)R(1)M(2)B
5
Colony DLL:
bC----------BbCR-M--B
2
Please enter the building type:
m
The building of type m is not found in the colony.
2
Please enter the building type:
M
The building of type M has been deleted from the colony.
7
Stock DLL:
Biofuel(398)
Self-ReplicatingBots(15)
SolarPanels(721)
Hydrogel(221)
Nanomaterials(80)
3
Colony DLL:
bCBbCRB
(0)b(0)C(11)B(0)b(0)C(0)R(4)B
5
Colony DLL:
bC----------BbCR----B
7
Stock DLL:
Biofuel(398)
Self-ReplicatingBots(15)
SolarPanels(721)
Hydrogel(221)
Nanomaterials(80)
8
Clearing the memory and terminating the program.

Some Important Rules

In order to get full credit, your program must be efficient, modular (with the use of functions), well commented and properly indented. Besides, you also have to use understandable identifier names. Presence of any redundant computation, bad indentation, meaningless identifiers or missing/irrelevant comments may decrease your grade in case that we detect them. When we grade your THEs, we pay attention to these issues. Moreover, **we may test your programs with very large test cases**. Hence, take into consideration the efficiency of your algorithms other than correctness.

Sample runs give a good estimate of how correct your implementation is, however, we will test your programs with different test cases and **your final grade may conflict with what you have seen on CodeRunner**. We will also **manually** check your code, indentations and so on, hence do not object to your grade based on the **CodeRunner** results, but rather, consider every detail on this documentation. **So please make sure that you have read this documentation carefully and covered all possible cases, even some other cases you may not have seen on CodeRunner or the sample runs**. The cases that you *do not need* to consider are also given throughout this documentation.

Submit via CodeRunner on SUCourse ONLY! Paper, e-mail or any other methods are not acceptable.

The internal clock of SUCourse might be a couple of minutes skewed, so make sure you do not leave the submission to the last minute. In the case of failing to submit your THE on time:

"No successful submission on SUCourse on time = A grade of zero (0) directly."

What and where to submit (PLEASE READ, IMPORTANT)

It'd be a good idea to write your name and last name in the program (as a comment line of course). Do not use any Turkish characters anywhere in your code (not even in comment parts). If your full name is "Duygu Karaoğlan Altop", and if you want to write it as comment; then you must type it as follows:

// Duygu Karaoglan Altop

You should copy the full content of the .cpp file and paste it into the specified "Answer" area in the relevant assignment submission page on SUCourse. ***Please note that the warnings may be considered as errors on CodeRunner, which means that you should have a compiling and warning-free program***.

Since the grading process will be automatic, you are expected to strictly follow these guidelines. If you do not follow these guidelines, your grade will be zero (0). Any tiny change in the output format will result in your grade being zero (0), so please test your programs yourself, and against the sample runs that are available at the relevant assignment submission page on SUCourse.

In the CodeRunner, there are some visible and invisible (hidden) test cases. You will know whether your code has successfully passed all the test cases or not before submitting your code. There is no re-submission. You don't have to complete your task in one time, you can continue from where you left last time but you should not press submit 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.

You may visit the office hours if you have any questions regarding submissions.

How to get help?

You may ask your questions to TAs or to the instructor. Information regarding the office hours of the TAs and the instructor are available at SUCourse.

Plagiarism

Plagiarism is checked by automated tools, and we are very capable of detecting such cases. Be careful with that. Exchange of abstract ideas are totally okay but once you start sharing the code with each other, it is very probable to get caught by plagiarism. So, do NOT send any part of your code to your friends by any means or you might be charged as well, although you have done your THE by yourself. THEs are to be done personally and you have to submit your own work. **Cooperation will NOT be counted as an excuse.**
In case of plagiarism, the rules on the Syllabus apply.

Good Luck!
Ahmed Salem, Duygu Karaoğlan Altop