UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS132 - Imperative Programming

Practical 9 Specifications

Release Date: 12-05-2025 at 06:00

Due Date: 16-05-2025 at 23:59

Late Deadline: 18-05-2025 at 23:59

Total Marks: 112

Read the entire specification before starting with the practical.

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you begin coding.***
- This assignment should be completed individually.
- **Every submission will be inspected with the help of dedicated plagiarism detection software.**
- Be ready to upload your assignment well before the deadline. There is a late deadline which is 48 hours after the initial deadline which has a penalty of 20% of your achieved mark. **No extensions will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or structure).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use C++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software to generate submitted code is strictly forbidden and will be considered as plagiarism.

2 Overview

In this practical, you will gain experience with reading from a CSV file, as well as 1D dynamic arrays that contain dynamic values.

3 Background

In this practical, you will be creating a linear dynamic data structure that is used to store floats. The data structure (an example is provided in Figure 1) consists of arrays connected together to form a chain of blocks. To add additional complexity, each array contains pointers to dynamic floats. Thus, you will be implementing a chain of blocks, using `float**`.

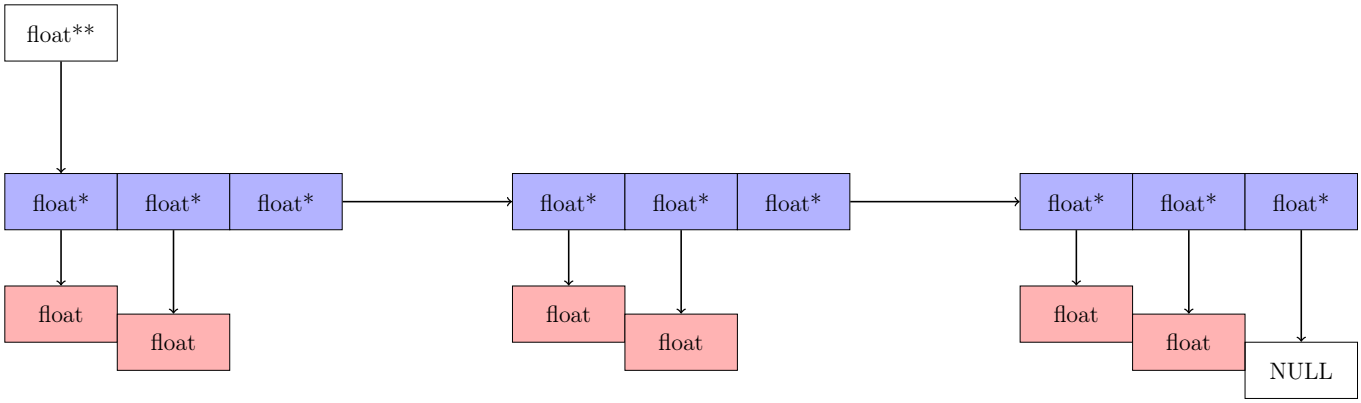

Figure 1: Example of the linear dynamically sized datastructure you will be implementing

In Figure 1, the blue blocks indicate arrays of type `float*`, which can be accessed by a pointer of type `float**`. A problem that now arises is how we can store a `float**` inside of a `float*`. You have been provided with two functions that can perform the conversion from `float*` to `float**` and vice versa. The red blocks indicate dynamic float values, but can also be NULL. In other words, the data structure can contain "gaps" where the `float*` points to NULL. It is important to note that the last cell in each array will always point to either the next array or to NULL. Thus, if an array has a size of 3, the first two indices can be populated with data, while the last holds the possible location of the next array. For the remainder of the specification, a single array in the chain of arrays will be referred to to as a *block*.

4 Your Task:

You are required to implement all of the functions laid out in this section.

Task	**Mark**
insert	16
remove	18
countingFunctions	20
toArraySortRepack	33
testCSVSearchGet	14
Testing	11

4.1 blockChain

This namespace contains all of the functions and the constant variable that are used to create the described data structure.

4.1.1 Const variable

There is a single const variable, `const int blocksize`, which determines the size of the blocks used in the described data structure.

4.1.2 Functions

This section describes the functions you are required to implement. It is important to realise that the majority of the functions described below can be called on any block in the data structure and not just the first block. This allows you to exploit some recursive algorithms to simplify your implementations.

- `createBlock`
  - This function needs to create a block of the sized defined by the `blockSize` variable.
  - You are then required to initialise each index in the block to point to `NULL`.
- `insert`
  - This function inserts the passed in `value` into the first open "gap" in the data structure.
  - The `block` parameter indicates the current block.
  - *Hint: remember you will need to allocate memory to store the value in.*
  - If the `block` parameter is NULL, you need to create a new block and insert the value into the newly created block.
    - *Hint: remember to assign the newly created block to the passed in `block` parameter, if you wish to use the block outside of the function.*
  - If the passed-in block is "full", attempt to insert the value into the next block in the data structure.
  - Continue trying to insert the value until a gap has been found, or until the end of the data structure is reached.
    - If the end has been reached, then you will need to add a new block to the end of the data structure and insert the value into that block.
  - This function needs to return the position where the value was inserted. Remember that the first index in the data structure starts at 0, and that you should not count the "chaining" indexes.
  - Example:
    - Given the data structure in Figure 2, if we were to insert a value at the yellow position, the function would return 0.
    - Given the data structure in Figure 2, if we were to insert a value at the green position, the function would return 3.

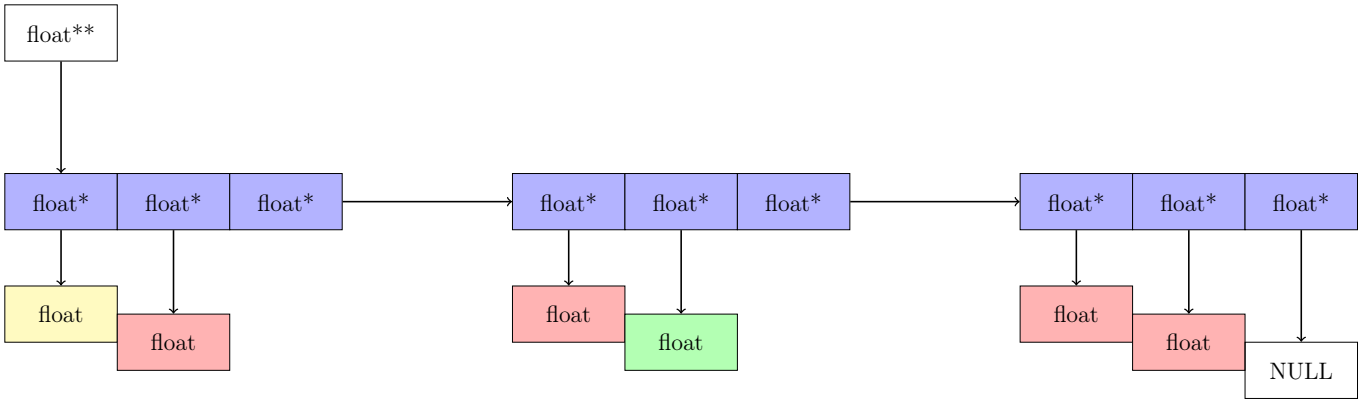

Figure 2: Insert Example

- `remove`
  - This function needs to remove all occurrences of the passed-in `value`, starting with the passed-in `block`.
  - Thus, you will need to iterate through the entire data structure and determine if each index contains the value or not.
  - If the index does contain the value, you will need to remove that value and set the index to `NULL`.
  - *Hint: remember to deallocate any dynamically allocated memory.*
  - The function needs to return the number of values that were removed.
  - After removing all occurrences of the value and a block becomes "empty", do not destroy it.
  - If the passed in `block` is `NULL`, return 0.
- `sort`
  - This function needs to sort the values in ascending order (smallest to biggest), with all the gaps being at the end of the data structure.
  - It can be assumed that the passed-in block is the first block in the data structure.
  - After sorting, if any blocks become "empty", do not destroy them.
  - Example:
    Assume that Figure 3 is unsorted, then Figure 4 is the result after sorting.

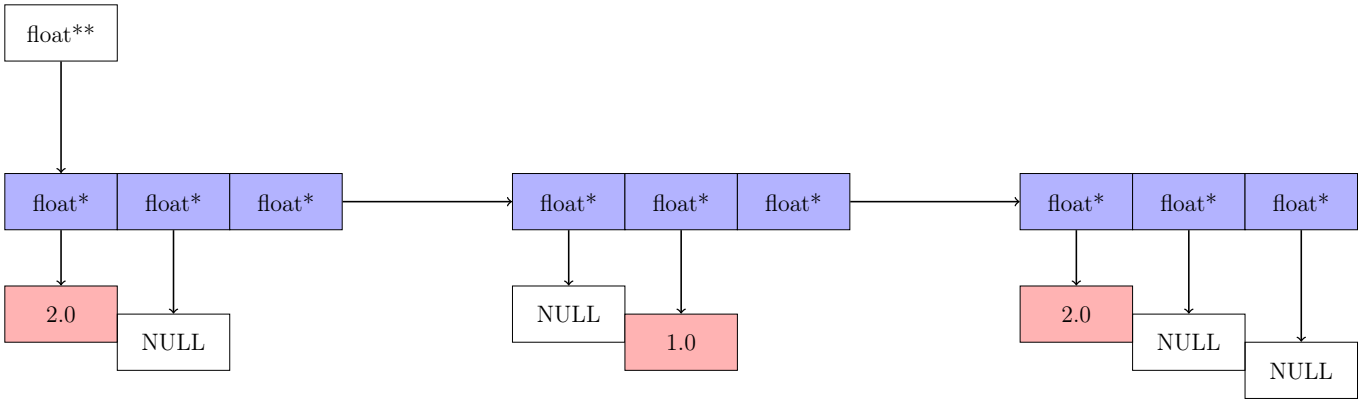

Figure 3: Unsorted Example

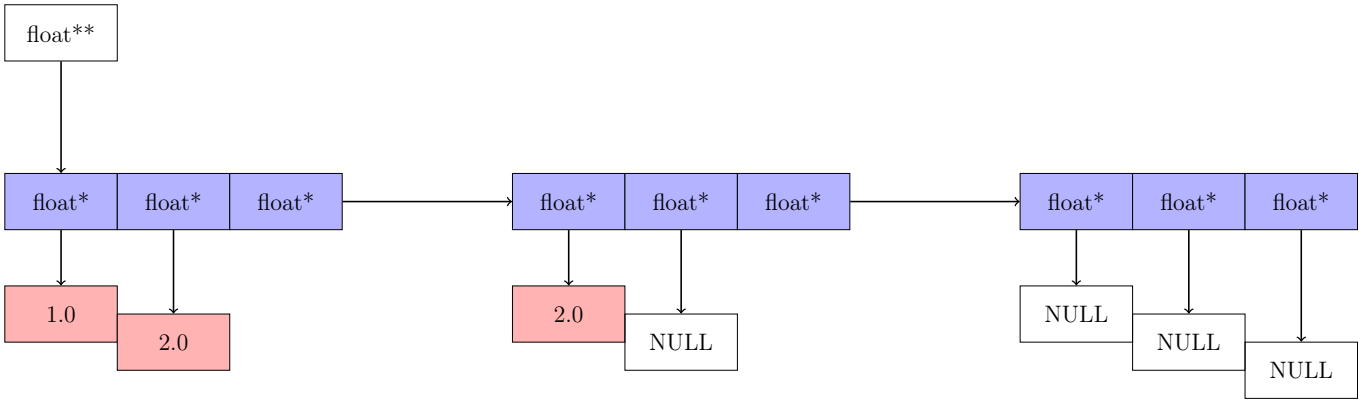

Figure 4: Sorted Example

- `toArray`
  - This function needs to "combine" all of the blocks into a single big array that contains no gaps, and return this array.
  - Assume that the passed-in `block` is the start of the data structure.
  - If the `block` is `NULL`, the function should return `NULL`.
  - *Hint: A function discussed later, can be useful to determine the size of the resulting array*
  - Example:
    - Consider `toArray` is called on the data structure illustrated in Figure 3, the resulting array would be: `[2,1,2]`

- `destroy`
  - This function is used to "clean-up" all empty blocks in the data structure.
  - Starting with the passed-in `block`, if a block is empty, it needs to be destroyed.
  - The trick comes in that you need to remember to re-chain the data structure.
  - *Hint: it is easier to destroy and re-chain the next block than it is to destroy and re-chain the current block.*
  - *Hint: You will thus need to account for 3 cases:*
    1. *An empty block is at the front of the data structure.*
    2. *An empty block is in the middle of the data structure.*
    3. *An empty block is at the end of the data structure.*
  - *Hint: Algorithms developed for linked list removal can be useful, and the following visual resource can be used to create your adapted algorithm:* `https://visualgo.net/en/list`
  - *Hint: Remember if the front block is removed, that the passed-in* `block` *is updated to point to the first "non-empty" block.*
  - *Hint: Remember that the end of the data structure should always point to* `NULL`*.*
- `isEmpty`
  - This function is used to determine if the data structure starting with the passed in `block` is empty or not.
  - *Hint: Remember empty is defined as no values stored in the data structure.*
  - *Hint: Remember if a block is* `NULL` *it is also seen as empty.*
- `repack`
  - This function is similar to the `sort` function, with the only difference that the values are not sorted, and empty blocks are removed.
  - In other words, this function needs to move all of the elements in the data structure such that all the gaps are at the end of the data structure.
  - Do not change the order of the values.
  - Example:
    Taking Figure 3 as the initial state of the data structure, Figure 5 shows the resulting data structure after the `repack` function was called.

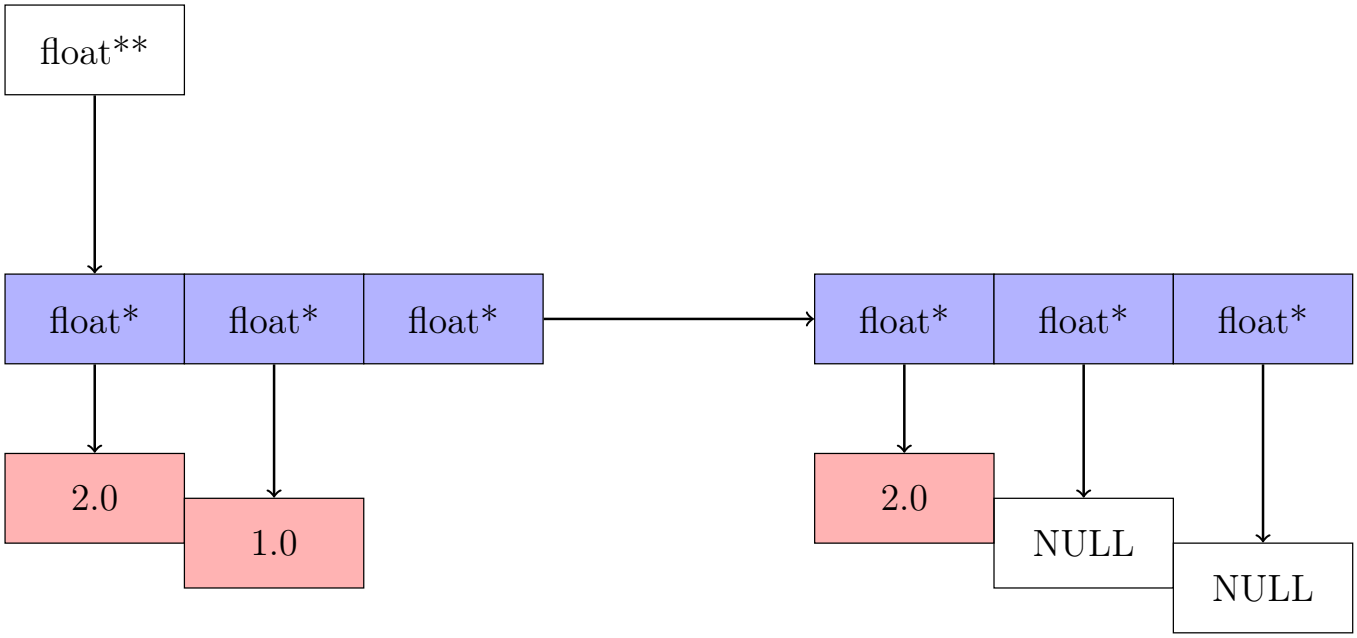

Figure 5: Repacked Example

- `search`
  - This function should determine if the passed-in `value` is contained in the data structure or not.
  - The passed-in `block` denotes the start of the data structure.
  - If the value is not in the data structure or the `block` is `NULL`, return false, else return true.
- `get`
  - This function needs to return the `float*` at the passed-in `position` in the data structure.
  - Assume the passed-in `block` is the start of the data structure.
  - *Hint: remember that the first index starts at 0, and do not count the chaining indices.*
  - If the passed-in parameters are invalid the function should return `NULL`.
- `numberOfBlocks`
  - This function needs to determine the number of `blocks` in the data structure.
  - If the passed-in `block`, which denotes the start of the data structure, is `NULL`, return 0.
  - Example: The number of blocks in Figure 3 is 3, and in Figure 5 is 2.
- `numberOfFloats`
  - This function needs to determine how many float values are contained in the data structure.
  - If the passed-in `block`, which denotes the start of the data structure, is `NULL`, return 0.
  - Example: The number of floats in Figure 3 is 3.

- `maxPossibleFloats`
  - This function needs to determine, given the number of blocks present in the data structure and the size of the blocks, what the theoretical maximum number of floats are that the data structure can contain.
  - If the passed-in `block`, which denotes the start of the data structure, is `NULL`, return 0.
  - Example: The number of blocks in Figure 3 is 6, and in Figure 5 is 4.
- `total`
  - This function needs to calculate the sum of all the values in the data structure.
  - If the passed-in `block`, which denotes the start of the data structure, is `NULL`, return 0.
- `average`
  - This function needs to calculate the **average** of all the values in the data structure.
  - If the passed-in `block`, which denotes the start of the data structure, is `NULL` or the data structure is empty, return 0.
- `adjust`
  - This function needs adjust (add) each float value in the passed-in data structure with the passed-in `value`.
  - If the passed-in `block`, which denotes the start of the data structure, is `NULL` or the data structure is empty, the function should not modify the data structure.
- `loadFromFile`
  - This function needs to create and populate a data structure from the contents of a CSV file, specified by the passed-in `csvFileName`.
  - It can be assumed that the file exists.
  - You will need to read each value in the CSV file and insert them into the data structure.
  - There is no limit to the amount of values on a line or the amount of lines in the file.
  - It can however be assumed that each line of the file does contain at least a single float value.
  - Example of possible files that will create a data structure like Figure 5
    1. FileA

```
2.0,1.0,2.0
```

    2. FileB

```
2.0,1.0
2.0
```

3. FileC

```
2.0
1.0,2.0
```

4. FileD

```
2.0
1.0
2.0
```

4.2 providedFiles

This namespace is provided for you and should not be modified. The purpose of the functions are to perform the conversion from `float*` to `float**` and visa versa.

5 Memory Management

As this practical has a heavy memory management focus, specialised tools will be used to evaluate you on your memory management skills.

The tool that will be used is `valgrind`. To determine the effectiveness of your memory management skills, the following command can be used:

```
valgrind --leak-check=full --keep-stacktraces=alloc-and-free ./main
```

This command assumes that the code was compiled to an executable `main`, and that the debugging flag was used during compilation.

6 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

The mark you will receive for the testing coverage task is determined using Table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

7 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **C++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- Do not include using `namespace std` in any of the files.
- You may only use the following libraries:
  - string
  - fstream
  - sstream

8 Upload Checklist

The following C++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- blockChain.cpp
- `main.cpp`
- Any text/csv files used by your `main.cpp`

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

9 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ -std=C++98 -Wall -Werror -g *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 10 submissions and your final submission's mark will be your final mark. Upload your archive to the Practical 9 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**