COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 5 Deque

Page maintained by

DINH, Anh Dung
Email: dzung@ust.hk
Last Modified: 10/13/2022 15:20:45

Homepage

Course Homepage

Honor Code

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

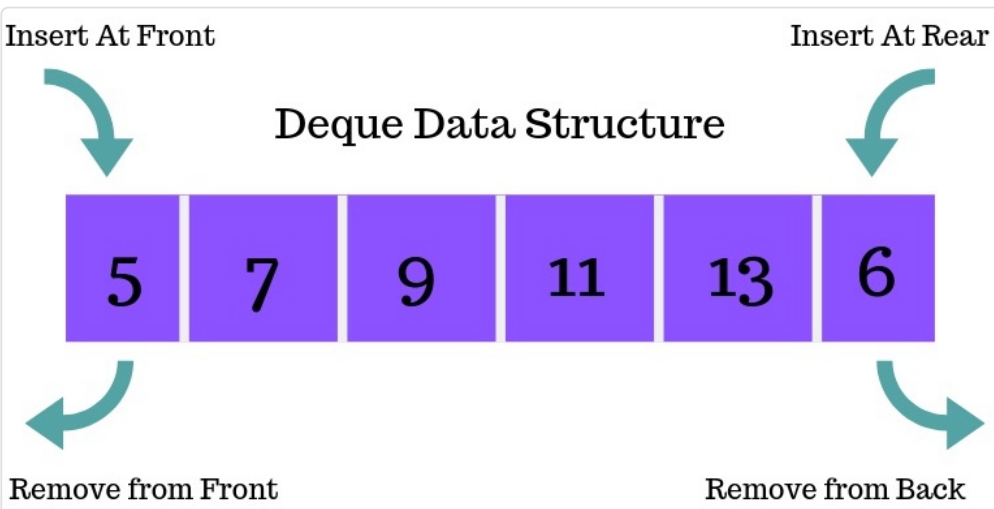
End of Honor Code

Objectives & Intended Learning Outcomes

The objective of this assignment is to provide you with practice on structs, pointers, linked lists and file IO. Upon completion of this assignment, you should be able to:

1. Use pointers to manipulate data
2. Implement a linked list and its variants
3. Use structs to define new data types.
4. Use File IO to read from / write to files.

End of Objectives & Intended Learning Outcomes



"Deque" is a special data structure, where elements can be added to or removed from either the front (head) or back (tail).

Source: https://learnersbucket.com/tutorials/data-structures/implement-deque-data-structure-in-javascript/

Introduction

The goal of this assignment is to implement a "deque", which is a special data structure, with the help of "circular doubly-linked list with a sentinel node", a variant of the linked list we saw during class.

A "double-ended queue", or "deque" (pronounced deck) is a special data structure, for which

elements can be added to or removed from either the front (head) or back (tail).

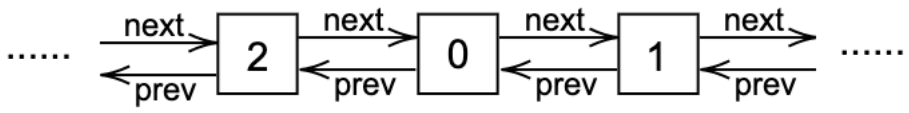
Circular doubly-linked list with a sentinel node

Don't worry about the long name, let's break it down step by step.

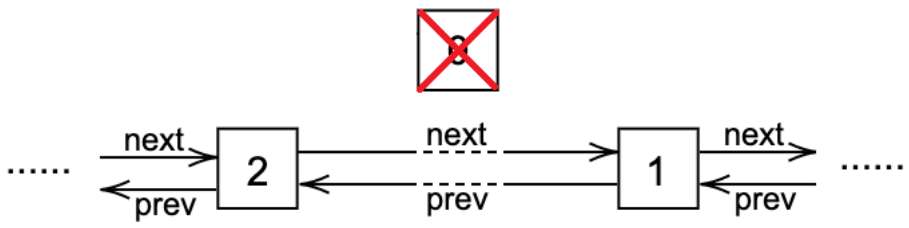
Doubly-linked list

In the singly-linked list, "delete" is not a handy stuff. Whenever you want to delete a node, you need to first delete it, and then connect its previous node with its next node. However, we cannot visit its previous node from itself, so we need to iterate though the list and always keep

track of the previous node of current node.



To solve this problem, we can connect two nodes in both ways: if `node_x.next = node_y`, then we also have `node_y.prev = node_x`. Now, we can access the previous node from the node itself, and when we delete node `curr`, we can simply do `curr.prev.next = curr.next`, which means "for the previous node of the deleted node, its `next` pointer will point to the next node of the deleted node now"; we also need to update "the previous node of the next node", i.e., `curr.next.prev = curr.prev`. (Note: special cases will be handled below, when the list only has 0 or 1 items)



This frees us from always keeping track of the previous node of current node. Usually in doubly-linked list, we will store both first node and last node of the list, so that we can iterate either from first to last or from last to first backwards.

Sentinel node

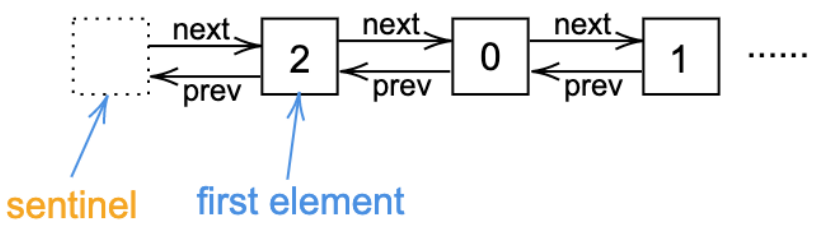
In the singly-linked list taught during lectures, you may have already noticed something annoying: *when we insert or delete an item, we ALWAYS need to check some special cases, e.g., if the list is empty, or if the list contains only one element.* The **doubly-linked list** alone will not solve the problem. For example, when you want to delete an element in the list, you need to consider:

1. When the element is the first element
2. When the element is the last element
3. Other cases

Example implementation in code:

```
if (curr -> prev != nullptr) curr -> prev -> next = curr -> next;
        else first = curr -> next;
if (curr -> next != nullptr) curr -> next -> prev = curr -> prev;
        else last = curr -> prev;
```

To avoid having to check for different cases each time we perform list operation, we can put a dummy **"sentinel node"** at the beginning of the list, no matter whether the list is empty or not:

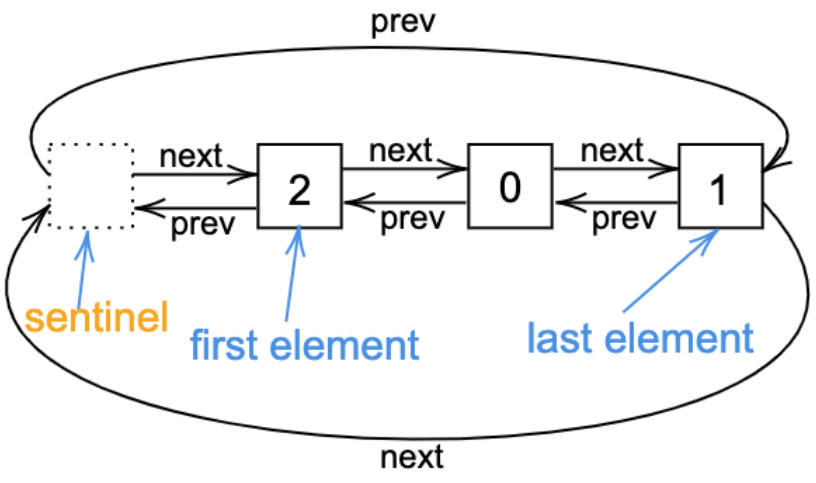


Now the code becomes:

```
curr -> prev -> next = curr -> next;
if (curr -> next != nullptr) curr -> next -> prev = curr -> prev;
        else last = curr -> prev;
```

Circular list with a sentinel

The sentinel node is never selected for deletion, so we no longer need to check if the deleted node is at the beginning of the list. This simplifies the number of cases, but we still need to check if the node to delete is the last element. A solution is to add another sentinel node at the end; however, it can be more efficient to reuse the existing sentinel node:



Since the last element is connected with the sentinel node, we say the list is **circular**. Then there is no need to worry whether the element we are going to delete is the first or last: they will never be! The code now becomes:

```
curr -> prev -> next = curr -> next;
curr -> next -> prev = curr -> prev;
```

That's a **circular doubly-linked list with a sentinel**. This kind of list should be very simple to implement because there are no special cases to consider!

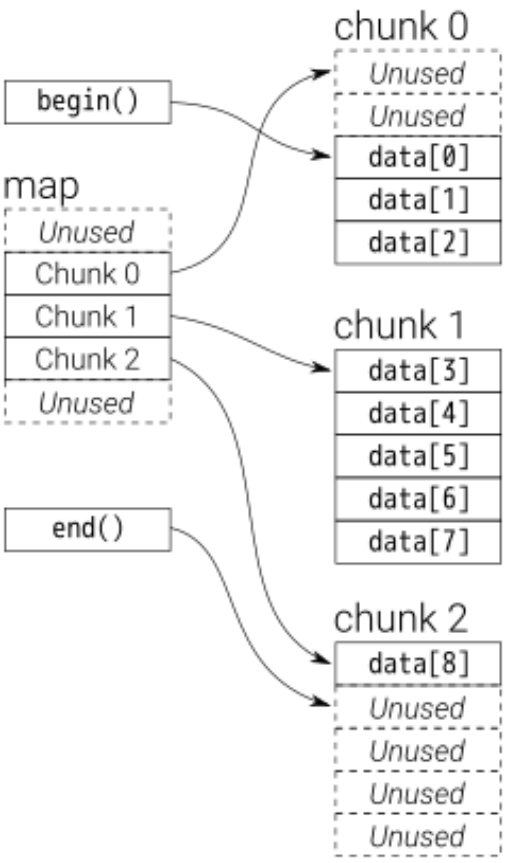
End of Introduction

Description

Please read the FAQ section for some common clarifications. You should check that a day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by then.

How does a deque look like?

In C++ STL (Standard Template Library), deque is provided as a standard container in `#include <deque>`. It is implemented using "a bunch of arrays with fixed size".



Source: https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl

They do this because this structure supports very fast "random access". For example, if we want to get the 30-th element in the deque, we can find out which chunk to look at, and which position the element locates in the chunk *by computation*, instead of iterating through 30 items from the beginning.

However, this is relatively difficult to implement and manipulate. In this assignment, we will **replace the "map" shown in the figure above, with a "circular doubly-linked list with a sentinel", where each node in the list is a fixed-size array.**

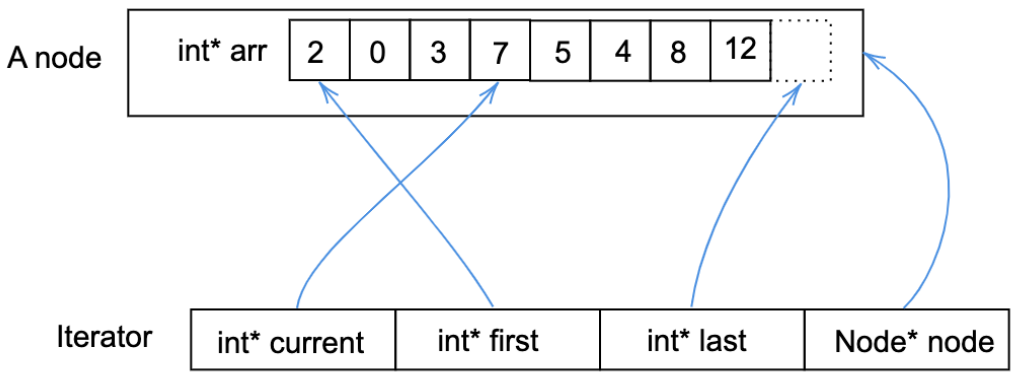
Iterator

In C++ STL, an **iterator** is used to point to the memory address of the container (like deque). For better understanding, you can relate them with a pointer, to some extent.

Why do we need the iterator? Consider the following case: when you now points at the last item in a node, and you want to move on to next item, you need to move to next node in your list. However, it is not possible with only a pointer: you don't know which node you are currently at, not to mention where is your next node!

To solve that situation, we **wrap the pointer and which node we are currently at**, into a structure. To be more convenient later, we will record the first and last address of this node, and also keep them in the structure. That's an **iterator**, and that's exactly what C++ does in its STL.

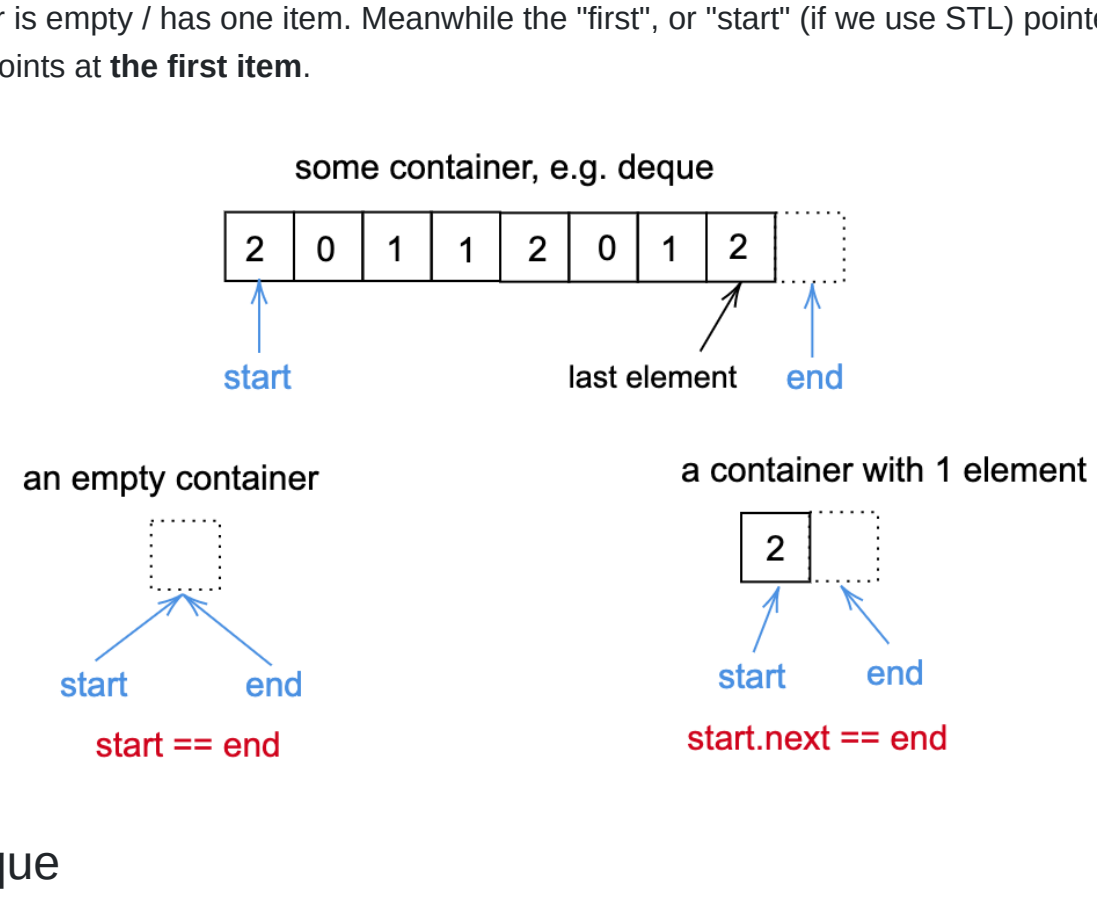
When an iterator points to item 7:



```
struct Node {
    int arr[CHUNK_SIZE];        // the chunk
    Node* prev;                 // previous Node
    Node* next;                 // next Node
};

struct Iterator {
    int* current;           // current position
    int* first;             // begin of the chunk, i.e., the position of first item
    int* last;              // end of the chunk, i.e., the position after the last item
    const Node* node;       // current Node
};
```

Back to the situation we discussed above, when we need to move on to next node, we will have:



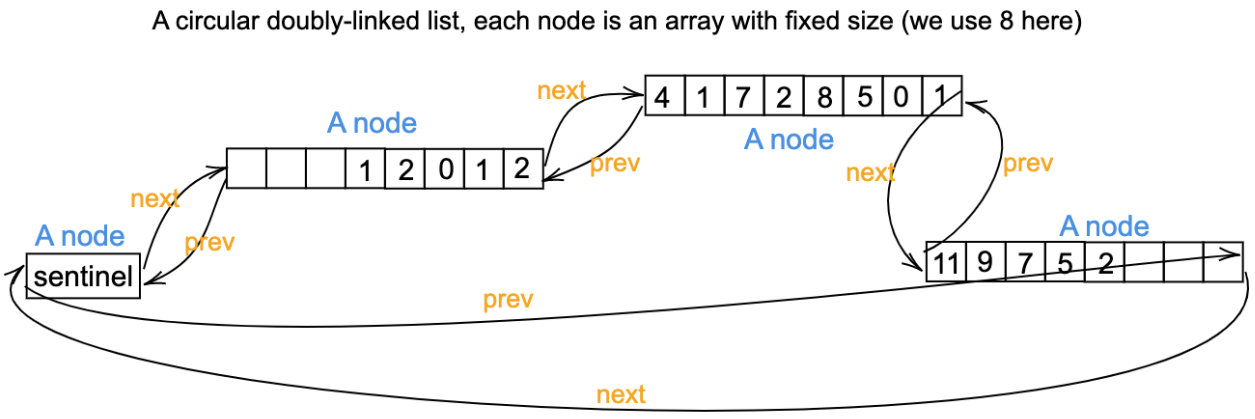
You may notice that the "last", or "end" (if we use STL) pointer in the iterator always points at **the position next to the last item**. This is designed in such a way to distinguish when a container is empty / has one item. Meanwhile the "first", or "start" (if we use STL) pointer always points at **the first item**.

A Deque

Now, what we need in a deque is (1) **two iterators**, representing the position of first item and the position next to the last item in the deque, as well as (2) **a circular doubly-linked list with sentinel**, where each node is an array of fixed size. We use 8 in this assignment (stored in a const variable `CHUNK_SIZE`).

```
struct Deque {
    Iterator start;     // the position of first item in deque
    Iterator end;       // the position after last item in deque

    Node* sentinel;     // sentinel of the circular doubly-linked list
    int ll_size;        // size of linked list, this is number of chunks, not items
};
```



In this assignment, you are required to:

- Implement the basic functions of a deque **Iterator**.
- Implement the basic functions of a **deque**.

Additional Remarks

- You are required to implement
  - All part I tasks in `Deque_Iterator.cpp`,
  - All part II tasks in `Deque.cpp`,

  You need to create these two files on your own, and **submit only these two files to ZINC for grading**.
- You are allowed to define your own helper functions, but be careful as ZINC won't be able to know of their existence. This shouldn't be a problem as long as the tasks' function return values are as expected.
- All the task functions are in global scope, and you are allowed to have the task functions call each other for easier implementation. Be careful when two or more functions call each other reciprocally, as they may enter an infinite loop.
- You are only allowed to use two header files `#include <iostream>` and `#include <fstream>` for File IO, in addition to the two header files `Deque_Iterator.h` and `Deque.h` we provided.
- **You are strictly forbidden to modify the two header files we provide,**. i.e., `Deque_Iterator.h` and `Deque.h`. However, you can overload the functions in them. When we grade your program, if we cannot find the exact function whose signature is what we given in the header files, you will have **no score** for that task. This situation is unlikely to occur if you never modified the `main` function and if it runs well.

End of Description

Tasks

This section describes the functions that you will need to implement. Please refer to the `main` function given in skeleton to see how they are invoked.

Part I: Deque Iterator

Task 1 - Implement the `equal()` and `value()` functions

```
bool equal(const Iterator& it1, const Iterator& it2);
int value(const Iterator& it);
```

The `equal()` function checks whether the two iterators point to the same address, i.e., whether their current positions are the same. If they point to the same address, we consider they are equal, and return `true`; otherwise, return `false`.

The `value()` function returns the value that the iterator is currently pointing at. It should be a value in the deque.

You don't need any validity checking. We assure the iterators passed in are always well-initialized and point to meaningful addresses during grading. However, if you use them in other tasks later, make sure you pass valid parameters.

Task 2 - Move the Iterators

```
void set_node(Iterator& it, Node* new_node);
Iterator next(const Iterator& it);
Iterator prev(const Iterator& it);
```

The `set_node()` function will let the iterator `it` point at node `new_node`. Note that you should not update `it.current`, since this pointer is not determined by the node, while other variables in the iterator should be determined by the new node.

The `next()` function returns an iterator that points to the position next to the given `it`. You should **not** modify the `it` passed in, you need to create a new iterator and return it, so we passed `it` by `const` to avoid modifying.

The `prev()` function is similar to `next()`, but returns an iterator that points to the position before the given `it`.

You don't need any validity checking. We assure the iterators passed in are always well-initialized and point to meaningful addresses during grading. However, if you use them in other tasks later, make sure you pass valid parameters.

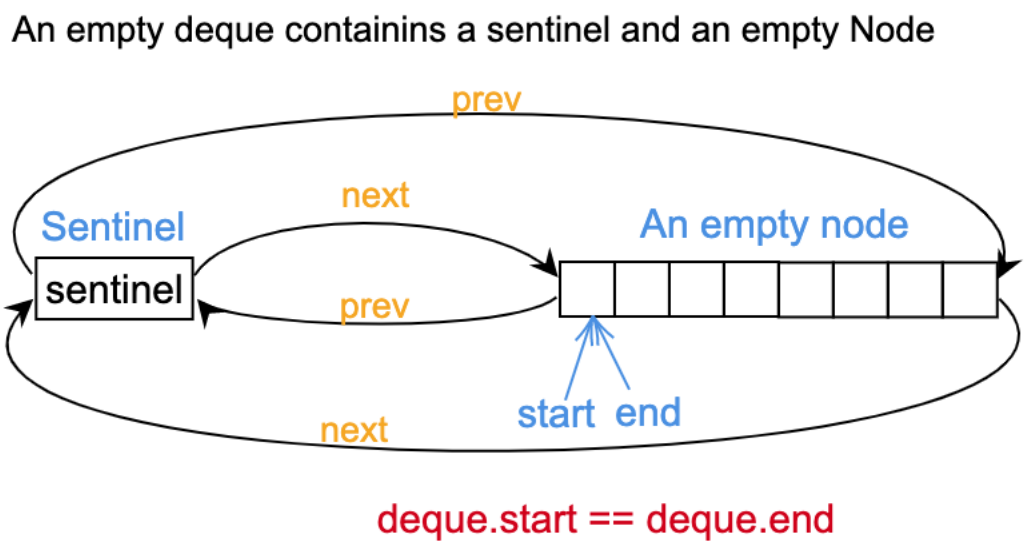
Note for Part I: The sample `main()` function we provided (i.e., the sample input/output) does not directly call those functions in Part I (other than `prev` being used for one function call). It is mainly used to check the correctness of your deque in Part II; nevertheless, many Part II functions require Part I implementations to work properly. However, please bear in mind that we **will directly call those functions in Part I during actual grading.**

Part II: Deque

Task 3 - Create and destroy the deque

```
Deque create_deque();
void destroy_deque(Deque& deque);
```

The `create_deque()` function returns an empty deque containing no elements. However, the linked list should contain **one sentinel node** and **at least one normal node with no element in its array**, so that the `start` and `end` Iterator of the deque both point at **some position of the array in a normal node**. (Recall that they should point at same position for an empty deque, but for simplicity you don't have to let them both point at the first position as the image shown below)



The `destroy_deque()` function destroys the deque given. It should deallocated all memories allocated for this deque, if any, and also set `sentinel` to a `nullptr`. After the deque is destroyed, the deque becomes "empty" and should never be used again.

Task 4 - Front and back of the deque

```
Iterator begin(const Deque& deque);
Iterator end(const Deque& deque);
int front(const Deque& deque);
int back(const Deque& deque);
```

The `begin()` and `end()` function simply return iterators that represent the start and end iterator of the deque. Recall that `end()` should point to the position next to the last element in the deque. (hence the address may not belong to the deque, that's ok)

The `front()` and `back()` function returns the value of first and last element in the deque. If the deque is empty, you should print

```
cout << "Cannot get front: deque is empty" << endl;
// or
cout << "Cannot get back: deque is empty" << endl;
```

and we will not care about the return value, you may just return `-1` or any unused integer. *(Additional note: in STL, get the first/last element in an empty deque will raise an error. In this PA, we simply do this because we haven't learned how to raise an error)*

Task 5 - Size of the deque

```
bool empty(const Deque& deque);
int size(const Deque& deque);
```

As indicated by their names, `empty()` returns `true` if the deque is empty, and `false` otherwise; while `size()` returns the number of elements in the deque. *Hint:* you don't need to iterate through the deque to get this value, you can compute it since each node has a fixed size of array.

Task 6 - Operate the deque at front or back

```
void push_back(Deque& deque, int val);
void push_front(Deque& deque, int val);
void pop_back(Deque& deque);
void pop_front(Deque& deque);
```

The `push_back()` function inserts the `val` as the last element in the deque, while `push_front()` function inserts the `val` as the first element.

On the other hand, the `pop_back()` function removes the last element in the deque while `pop_front()` removes the first element. If the deque is empty, you should print

```
cout << "Cannot pop_back: deque is empty" << endl;
// or
cout << "Cannot pop_front: deque is empty" << endl;
```

and the function does nothing other than printing the message. *(Additional note: in STL, pop from an empty deque will raise an error. In this PA, we simply do this because we haven't learned how to raise an error)*

These four operations are basic operations for a deque, which is where the name of deque (double-ended queue) comes from.

Hint: You may need to create a new node or delete an existing node when you insert or delete an element.

Hint: Values outside the `deque.start` and `deque.end` range are treated as garbage values and would be overwritten if new values are added at their positions. In other words, you don't need to worry about deleting/setting the `int` values to 0 when removing an element from the deque.

Task 7 - Print the deque

```
void print_deque(const Deque& deque);
```

This function prints the whole deque, in the format like `[2, 1, 3]` (you may see examples in sample output). If the deque is empty, simply print `[]`. The print should end with an `endl` (see sample output).

Hint: You may use iterator to walk through the deque, from start to end.

Task 8 - Insert into / erase from the deque

```
void insert(Deque& deque, const Iterator& pos, int val);
void erase(Deque& deque, const Iterator& pos);
```

The `insert()` function insert the given value `val` into the deque, before the **item** indicated by `pos`. For example, if `pos` points to the first item in deque, then we insert `val` before the first item, i.e., `val` will be the first item in the new deque.

The `erase()` function remove the value at the position indicated by `pos`. After removal, elements after `pos` should be moved forward, i.e., each array should be continuous, there should not exist a "blank position". For example, if we remove 3 in `[2, 3, 1]`, it should become `[2, 1]` rather than `[2, _ , 1]`

During actual grading, the `pos` iterator will always be valid. For example, we will never call `erase()` on `deque.end` iterator, which points outside the deque.

Additional note: Though we usually operate at the beginning or end of a deque, these two operations are still supported by the deque in STL. However, instead of using indices, we use iterators to represent positions. This will be much easier to implement.

Hint: You may need to create a new node or delete an existing node when you insert or delete an element.

Task 9 - Serialize the deque

```
void store_deque(const Deque& deque, const char* filename);
Deque load_deque(const char* filename);
```

In most computer programs, if you close the software, or even shutdown your computer, your data will still be kept after you restart the software. This is usually done by writing your data into a file and then re-load the data from the file later. If an object (like deque here) supports such operations, we say it is **serializable**.

The `store_deque()` stores all information of the given deque in the file identified by `filename`. Meanwhile, the `load_deque()` function loads the deque stored in file `filename`.

You may write anything to the file `filename`, as long as you are able to load deque from that file later. During the grading process when we do `load_deque`, the file `filename` always exists, and is always written by your program via `store_deque()`.

You don't need to ensure the **structure of the deque** loaded from file is exactly the same as the one you stored. As long as their contents are the same, that will be fine.

End of Tasks

Resources & Sample I/O

- Skeleton code: download here
- Demo programs: We don't provide demo programs for this PA, since deque is an "Abstract Data Type (ADT)". The `main.cpp` should have provided a good reference on how to invoke each functions you implemented and how to test them. They are quite similar to what we will do later in actual grading.
- Sample program outputs: you should get the following outputs by directly run the `main.cpp` we provided without any modification. Here is an online file compare tool.

```
===== Test create and destroy deque =====
Deque created
The deque now is: []
Deque destroyed
===== End Test =====

===== Test push back =====
The deque now is: [10, 2012, 2022]
===== End Test =====

===== Test push front =====
The deque now is: [2022, 2012, 10]
===== End Test =====

===== Test get front and back =====
Cannot get front: deque is empty
Cannot get back: deque is empty
front: 10, back: 10
front: 10, back: 15
front: 20, back: 15
===== End Test =====

===== Test empty and size =====
deque is empty: true
size of deque: 0
deque is empty: false
size of deque: 1
deque is empty: false
size of deque: 2
deque is empty: false
size of deque: 3
===== End Test =====

===== Test pop back =====
Cannot pop_back: deque is empty
The deque now is: [20, 10, 15]
The deque now is: [20, 10]
The deque now is: [20]
===== End Test =====

===== Test pop front =====
Cannot pop_front: deque is empty
The deque now is: [20, 10, 15]
The deque now is: [10, 15]
The deque now is: [15]
===== End Test =====

===== Test insert =====
The deque now is: [10, 15, 25]
The deque now is: [5, 10, 15, 25]
The deque now is: [5, 10, 15, 20, 25]
===== End Test =====

===== Test erase =====
The deque now is: [10, 15, 25]
The deque now is: [15, 25]
The deque now is: [15]
===== End Test =====

===== Test store and load deque =====
The deque now is: [10, 15, 25]
Successfully stored deque to 'deque_demo.txt'
The restored deque is: [10, 15, 25]
===== End Test =====
```

End of Resources & Sample I/O

Submission & Grading

Deadline: Sat, 15/10/22 HKT 23:59.
Compress the two source code files `Deque_Iterator.cpp` and `Deque.cpp` as `PA5.zip` for submission to the ZINC Autograding System. The ZINC submisssion portal for PA3 will be available soon after the announcement of this assignment.

Late Policy:
Please refer to the 3-day late budget policy.

The `main.cpp` we given in skeleton will be replaced when we grade your program. **You are strictly forbidden from modifying the given header files**. However, you can overload the functions in them. When we grade your program, you should not submit the header files, and we will use the ones provided in skeleton code. If we cannot find the exact functions whose signature is given in the two skeleton header files, you will have **no score** for that task. This situation is unlikely to occur if you do not modify `main.cpp` and it runs without error.

Before deadline, the test cases on ZINC will only contain a few cases. If you passed those tests, it only means your program can successfully run on ZINC and can pass those test cases. The result is by no mean complete and the score are irrelevant to your actual score. In actual grading stage after deadline, we will use a **different set of test cases**, which is expected to be more complete and more strict, to test the correctness your program. Your score will be determined based on the number of test cases you passed.

Please bear in mind that we will also check **memory leak** in test cases. If your program has memory leak for a test case, you may lose 50% of the score for that case.

Grading Scheme

The actual grading will use Unit Testing, i.e., we will test each functions individually, invoke the function you implemented, and check the return value/expected behavior of that function. The actual grading will also only be triggered, with the scores and test cases revealed, after the deadline. This hidden grading policy is for all the PAs in order to prevent reverse-engineering of the test cases, since the PAs are a significant part of the course assessment and course grade.

However, please bear in mind that for some functions, **we cannot test it without the help of other functions you implemented**. For example, if we want to test `print_deque()`, we must first push, or insert some items into the deque. This is only possible if we invoke your functions like `push_back()`, since each student may have a different deque structure and we cannot directly modify the inner structure of deque. If any test case fails because of another function you implemented is wrong, it is possible that you may submit an appeal and ask TA to manually check your implementation. More details will be provided after grading process.

We will execute unit testing on the task functions individually, and some test cases may include invalid input values. All situations for possible invalid inputs have been described in Tasks section. You don't have to check other special cases.

Please ensure that you submit to ZINC well before the deadline as **all late submissions will be automatically rejected**.

End of Submission & Grading

Frequently Asked Questions

Q: My code doesn't work, there is an error/bug, here is the code, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own and we should not finish the tasks for you. We are happy to help with explanations and advice, but we shall not directly debug the code for you.

Q: What are the restrictions regarding modifying the header files, writing our own helper functions, including extra header files, etc.?
A: The only hard restriction is that you can only submit `Deque_Iterator.cpp` and `Deque.cpp` to ZINC, and can only use two header files `#include <iostream>` and `#include <fstream>` in addition to the two header files `Deque_Iterator.h` and `Deque.h` we provided. Anything else that you do, while not strictly prohibited, **will be at your own risk regarding the PA5 grading result**. Please keep in mind that there is a grade penalty for all grade appeals that include modifications to your already submitted code (no matter how trivial the modification is).

Q: Am I allowed to use local function declarations (function declaration inside an existing function) for my helper functions?
A: You are strongly discouraged from doing so, as that "feature" is a leftover merely for backwards compatibility with C. In C++, it is superseded with class functions and lambda functions, which will be taught later in this course.

Q: I am confused about the `first`, `last`, `start` and `end` pointers.
A:

- `first` and `last` *pointers* are data members of **Iterator** that **point to the beginning and end of the Node's array**. Essentially, `first`, `last` and `node` should always be updated together to point to the Node containing the item being pointed at by `current`. They don't care about the Deque structure or how it is implemented (but you can assume the current Node's `next` and `prev` are pointing at valid Nodes).
- `start` and `end` *Iterators* are data members of **Deque** that **point to the start and end of the Deque's data range**. Their purpose is to tell the Deque which range of items are actually data of the Deque, and any items outside the range are treated as "garbage" values. These should be updated accordingly when the number of items in the Deque is changed via element insertion or deletion.

Q: When doing `insert()` and `erase()`, should we move the elements on the left or right of the inserted/deleted element?
A: Either is fine, as long as the Deque is printed out correctly. For example, if your current Deque looks like:

[_ _ _ _ 1 2 3 4] [5 6 7 8 _ _ _ _]

and item 5 is deleted, your implementation can cause the Deque to become either of these:

[_ _ _ _ 1 2 3 4] [6 7 8 _ _ _ _ _]
[_ _ _ _ _ 1 2 3] [4 6 7 8 _ _ _ _]

but as long as the Deque is printed as `[1, 2, 3, 4, 6, 7, 8]`, then you should get the point for relevant test cases.

Q: After adding the last item to a Node (for example, `push_back()` on a Deque so that the final Node's `arr[7]` is updated), should Iterator `end` be pointing at the position after `arr[7]` in the Node, or at the first item in the next Node (`node.next->arr[0]`)?
A: It should point at **the first item in the next Node** (you may need to make a new Node). Recall that `end` is an Iterator, so `end.current` can only point at positions `arr[0]` to `arr[7]`.

Q: Does `deque.ll_size` count the Sentinel node? For a newly initialized Deque, should `deque.ll_size` be 1 or 2?
A: Strictly speaking, `deque.ll_size` does not count the Sentinel node because it does not contribute to the Deque's items. So for a newly initialized Deque, `ll_size` is 1. However, this property is only for you to help maintain the Deque structure and assist in counting number of items, and not tested in any of the test cases. As long as you maintain this variable and use it correctly, there should be no problem.

Q: What does it mean by items having "garbage" values?
A: Whenever you initialize a Node with array of size 8, the 8 array elements might hold some garbage random int values, or set to some value upon initialization. But we don't say that an item is "deleted" when its value is 0, because it is still a valid integer value. The only thing determining whether an item is in the Deque or not is if its position is between the `start` and

`end` Iterators.

For more concrete example, suppose your Deque looks like this:

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ] [ 9, 10, 11, 12, 13, 14, 15, 16]
        ^                           ^
        start.current               end.current
```

where `deque.start.current` points at 3 and `end.start.current` points at 12. What this means is that the Deque values are from 3 to 11 (remember end points to the position after last element) - in other words, if you print out the Deque, it should be:

[3, 4, 5, 6, 7, 8, 9, 10, 11]

Even if the remaining array positions hold values 1, 2, 13, 14, 15, 16, they don't belong in the Deque, so we don't care what values they hold. Now, if you "delete" element 7 by simply changing its value, the new value just becomes a new element in the Deque. You need to maintain the `start` and `end` Iterators, such as:

```
[ 1, 2, 3, 4, 5, 6, 8, 9 ] [ 10, 11, 11, 12, 13, 14, 15, 16]
        ^                           ^
        start.current               end.current
```

Now, when printing the Deque, element 7 is truly deleted:

[3, 4, 5, 6, 8, 9, 10, 11]

End of FAQ

Change Log

17:30 30/9/22

- Fixed description of Task 2: You should **not** modify the `it` passed in...

19:30 07/10/22

- Added new FAQ questions to clarify some concepts.

14:30 09/10/22

- Added new FAQ question to clarify Iterator `end` behaviour.

18:15 10/10/22

- Added new FAQ question to clarify `ll_size` and item deletion.

15:20 13/10/22

- Fixed description of Task 8: The `insert()` function insert the given value `val` into the deque, before the ~~position~~ **item** indicated by `pos`.
- Clarification on the definition above: Suppose a Deque is [0, 1, 2, 3], and `pos` points to 2. Calling `insert(deque, pos, 4)` should modify the Deque to become [0, 1, 4, 2, 3].

End of Change Log