Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Basic Data Structure I • DIM0119
◁ Programming Project: Implementing the Deque Abstract Data Type ▷
30 de outubro de 2023

Overview

In this document we describe the *Deque* Abstract Data Type (ADT). We focus on two aspects of the deque: the core operations that should be supported, and how the data is stored and maintained internally. For this particular project, you should use several **dynamic arrays** as the data storage structure.

In the next sections we introduce the basic definition of terms related to a Deque ADT, its properties and operations, followed by coding aspects related to a Deque ADT implemented with a dynamic arrays.

Sumário

1 Definition of a Deque

We define a *general linear list* as the set of $n \geq 0$ elements $A[0], A[1], A[2], \ldots, A[n-1]$. We say that the size of the list is n and we call a list with size $n = 0$ an **empty list**. The structural properties of a list comes, exclusively, from the relative position of its elements:-

– if $n > 0$, $A[0]$ is the first element,
– for $0 < k \leq n$, the element $A[k-1]$ precedes $A[k]$.

Therefore, the first element of a list is $A[0]$ and the last element is $A[n-1]$. The **position** of element $A[i]$ in a list is i.

In addition to the structural properties just described, a list is also defined by the set of operations it supports. Typical list operations are to print its content; to clear the list; to access one element at a specific position within a list; to insert a new element at one of the list's ends, or at a specific location within the list; to remove one element at a given location within a list, or a range of elements; to inquire whether a list is empty or not; to get the size of the list, and so on.

The selection of which of these operations a list data structure needs to supports depends on their performance, in terms of time complexity. The **deque** (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both ends.

As opposed to a vector (dynamic array), the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays, with additional bookkeeping. Consequently, indexed access to deque must perform two pointer dereferences, in contrast with the single access of a vector.

As with the vector, the storage of a deque is automatically expanded and contracted as needed. Expansion of a deque is cheaper than the expansion of a vector because it does not involve copying of the existing elements to a new memory location. On the other hand, deques typically have large minimal memory cost because of the list of blocks it needs to maintain.

The complexity (efficiency) of common operations on deques is as follows:

- Random access - constant $O(1)$.
- Insertion or removal of elements at the end or beginning - constant $O(1)$.
- Insertion or removal of elements - linear $O(n)$.

In this document, in particular, we address how to implement a dynamic deque data structure. This versions of a list is equivalent to the `std::deque`. Later on, after learning how to implement *linked lists*, we will revisit the List ADT to explore alternative ways of implementing this ADT.

2 The List ADT

In this section we present the core set of operations a List ADT should support, regardless of the underlying data structure one may choose to implement a list with.

Most of the operations presented here and in the next sections follow the naming convention and behavior adopted by the STL containers.

2.1 Constructors, Destructors, and Assignment

Usually a class provides more than one type of constructor. Next you find a list of constructors that should be supported by our `deque` class.

Notice that the name of the class we want to implement is also `deque`. Therefore, to distinguish your `deque` from the STL's, we shall define our class within the namespace `sc` (a short for *sequence container*), which means the full name of the class shall be `sc::deque`.

In the following specifications, consider that `T` represents the template type.

Notice that all references to either a return type or variable declaration related to the size of a list are defined as `size_type`. This is basically an alias to some unsigned integral type, such as `long int`, `size_t`, for example. The use of an alias such as this is a good programming practice that enables better code maintenance. Typically we are going to define an alias at the beginning of our class definition with `typedef` or `using` keywords, as for instance in `using size_type = unsigned long`.

`deque( );`	(1)
`explicit deque( size_type count, const T& value );`	(2)
`explicit deque( size_type count );`	(3)
`template< typename InputIt >` `deque( InputIt first, InputIt last );`	(4)
`deque( const deque& other );`	(5)
`deque( std::initializer_list<T> ilist );`	(6)
`~deque( );`	(7)
`deque& operator=( const deque& other );`	(8)
`deque& operator=( std::initializer_list<T> ilist );`	(9)

(1) Default constructor that creates an empty container.
(2) Constructs the container with `count` copies of elements with value `value`.
(3) Constructs the container with `count` default-inserted instances of `T`.
(4) Constructs the container with the contents of the range `[first, last)`.
(5) Copy constructor. Constructs the container with the *deep copy* of `other`.
(6) Constructs the container with the contents of the initializer list `ilist`.
(7) Destructs the container. The destructors of the stored elements are called and the used storage is de-allocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.
(8) Copy assignment operator. Replaces the contents with a copy of the contents of other. This operation may require memory adjustments in case the capacity of both deques (i.e. `*this` and `other`) differ.
(9) Replaces the contents with those identified by initializer list `ilist`.

The two `operator=()` (overloaded) assign methods must return `*this` at the end, so we may have multiple assignments in a single command line, such as `a = b = c = d;`.

Parameters

`count` - the size of the list.
`value` - the value to initialize the list with.
`first, last` - the range to copy the elements from.
`other` - another list to be used as source to initialize the elements of the list with.
`ilist` - initializer list to initialize the elements of the list with.

2.2 Methods that do not require iterators

This set of methods comprises the operations that do not require iterators as input. The iterator-dependent operations will be introduced later, in Section 3.3, after we have properly introduced iterators.

- `size_type size() const`: return the number of elements in the container.
- `void clear()`: remove all elements from the container.
- `bool empty()`: returns `true` if the container contains no elements, and `false` otherwise.
- `void push_front( const T & value )`: adds `value` to the front of the list.
- `void push_back( const T & value )`: adds `value` to the end of the list.
- `void pop_back()`: removes the object at the end of the list.
- `void pop_front()`: removes the object at the front of the list.
- `const T & back() const`: returns the object at the end of the list.
- `const T & front() const`: returns the object at the beginning of the list.
- `void assign( sizet_type count, const T & value )`: replaces the content of the list with `count` copies of `value`.
- `T & operator[]( size_type pos)`: returns the object at the index `pos` in the array, with no bounds-checking.
- `T & at ( size_type pos )`: returns the object at the index `pos` in the array, with bounds-checking. If `pos` is not within the range of the list, an exception of type `std::out_of_range` is thrown.
- `size_type capacity() const`: return the internal storage capacity of the array.
- `void resize(size_type count)`: Change the number of elements stored in the container. If the current size (`size()`) is greater than `count`, the container is reduced to the first `count` element. On the other hand, if the current size is less than `count`, then additional *default-inserted* elements are appended to the container. Lastly, if current size is equal to `count`, nothing happens.
- `shrink_to_fit()`: Requests the removal of unused capacity. It is a non-binding request to reduce `capacity()` to `size()`. It depends on the implementation if the request is

fulfilled.

All existing iterators, including the past-the-end iterator, are potentially invalidated.

2.3 Operator overloading — non-member functions

These are some important binary operators that may be implemented as functions, rather than methods. They are:

- `bool operator==(const sc::deque& lhs, const sc::deque& rhs);`: Checks if the contents of `lhs` and `rhs` are equal, that is, whether `lhs.size() == rhs.size()` and each element in `lhs` compares equal with the element in `rhs` at the same position.
- `bool operator!=(const sc::deque& lhs, const sc::deque& rhs);`: Similar to the previous operator, but the opposite result.

In both cases, the type `T` stored in the list must be `EqualyComparable`, in other words, it must support the `operator==()`.

Note that these **are not methods nor friend functions**, but some plain-old regular functions. Therefore, the implementation of these functions must rely only on public methods provided by each class. This is an alternative way of providing operator overloading for classes. The `lhs` and `rhs` represent, respectively, the *left-hand-side* and the *right-hand-side* elements of a comparison operation.

3 Iterators

The next set of operations require the ability to insert/remove elements in the middle of the list. For that, we require the notion of *position*, which is implemented in STL as a nested type `iterator`.

Iterators may be defined informally as *a class that encapsulate a pointer to some element within the list*. This is an object oriented way of providing some degree of access to the list without exposing the internal components of the class, thus preserving the **encapsulation**[1] principle.

Before delving into more details on how to implement iterators, let us consider a simple example of iterators in use. Suppose we declared a doubly linked list of integers and wish to create an iterator variable `it` that points to the beginning of our list. Next, we wish to display in the terminal the content of the list separated by a blank space, and delete the first three elements of the list, after that. The code to achieve this would be something like:

```cpp
std::list<int> myList {1,2,3,4,5};  // List declaration.
// Let us instantiate an iterator that points to the beginning of the list.
std::list<int>::iterator it = myList.begin();
// Run through the list, accessing each element.
for ( ; it != myList.end(); ++it )
    std::cout << *it << " "; // Prints out: 1 2 3 4 5
```

[1]Besides **encapsulation**, the other fundamental principles of the Object-Oriented Programming paradigm are **inheritance**, **abstraction**, and **polymorphism**.

```
// Removing the first 3 elements from the list.
myList.erase( myList.begin(), myList.begin()+3 );
```

By examining this simple example we identify three issues we need to address if we wish to implement our own iterator object: *(i)* how one gets an iterator; *(ii)* what operations the iterators should support, and; *(iii)* which deque methods require iterators as parameters. These issues are discussed next and their corresponding methods depend upon the definition of the type `iterator` and `const_iterator`, which are nested classes of the `deque` class.

3.1 Getting an iterator

- `iterator begin()`: returns an iterator pointing to the first item in the deque (see Figure 1).
- `const_iterator cbegin() const`: returns a constant iterator pointing to the first item in the deque.
- `iterator end()`: returns an iterator pointing to the position just after the last element of the deque.
- `const_iterator cend() const`: returns a constant iterator pointing to the position just after the last element of the deque.

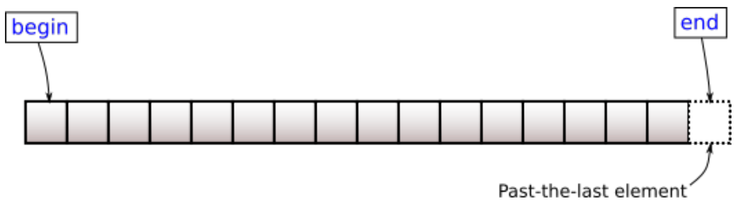

Figura 1: Visual interpretation of iterators in a container.
Source: http://upload.cppreference.com/mwiki/images/1/1b/range-begin-end.svg

The constant versions of the iterator are necessary whenever we need to use an iterator inside a `const` method, for instance. The `end()` method may seem a bit unusual, since it returns a pointer "out of bounds". However, this approach supports typical programming idiom to iterate along a container, as seen in the previous code example.

3.2 Iterator operations (random access iterator)

This is a list of operations that a random access iterator should support, according to the STL specification.

- `operator++()`: advances iterator to the next location within the deque. This signature corresponds to the prefix form, or `++it`.
- `operator++(int)`: advances iterator to the next location within the deque. This signature corresponds to the posfix form, or `it++`.

- `operator--()`: recedes iterator to the previous location within the deque. This signature corresponds to the prefix form, or `--it`.
- `operator--(int)`: recedes iterator to the previous location within the deque. This signature corresponds to the posfix form, or `it--`.
- `operator*()` as in `*it`: return a reference to the object located at the position pointed by the iterator. The reference may or may not be modifiable.
- `operator-()` as in `it1-it2`: return the difference between two iterators.
- `friend operator+(int n, iterator it)` as in `2+it`: return a iterator pointing to the `n`-th successor in the deque from `it`.
- `friend operator+(iterator it, int n)` as in `it+2`: return a iterator pointing to the `n`-th successor in the deque from `it`.
- `operator+=()` as in `it += n`: advances an iterator to the `n`-th successor in the deque from the location `it` originally pointed to.
- `operator-=()` as in `it -= n`: decrements an iterator to the `n`-th predecessor in the deque from the location `it` originally pointed to.
- `friend operator-(iterator it, int n)` as in `it-2`: return a iterator pointing to the `n`-th predecessor in the deque from `it`.
- `operator<()` as in `it1 < it2`: returns `true` if `it1` refers to a location in the deque that is located before the location pointed by `it2`.
- `operator>()` as in `it1 > it2`: returns `true` if `it1` refers to a location in the deque that is located after the location pointed by `it2`.
- `operator==()` as in `it1 == it2`: returns `true` if both iterators refer to the same location within the deque, and `false` otherwise.
- `operator!=()` as in `it1 != it2`: returns `true` if both iterators refer to a different location within the deque, and `false` otherwise.

Notice how theses operations involving iterators are (intentionally) very similar to the way we manipulate regular pointers to access, say, elements in an array.

3.3 Deque operations that require iterators

- `iterator insert( const_iterator pos, const T& value )`: adds `value` into the list *before* the position given by the iterator `pos`, after shifting to the right all the elements of the array starting at 'pos'. The method returns an iterator to the position of the inserted item.
- `iterator insert( const_iterator pos, size_type count, const T& value )`: inserts `count` copies of `value` into the deque *before* the position given by the iterator `pos`, after shifting to the right all the elements of the array starting at 'pos'. The method returns an iterator to the position of the inserted item.
- `template < typename InpuItr>`
  `iterator insert( const_iterator pos, InputItr first, InputItr last )`:

inserts elements from the range `[first; last)` before `pos`, after shifting to the right all the elements of the array starting at ‘pos‘.

- `iterator insert( const_iterator pos, std::initializer_list<T> ilist )`: inserts elements from the initializer list `ilist` before `pos`, after shifting to the right all the elements of the array starting at ‘pos‘. Initializer list supports the user of insert as in `myList.insert( pos, {1, 2, 3, 4} )`, which would insert the elements 1, 2, 3, and 4 in the list before `pos`, assuming that `myList` is a list of `int`.
- `iterator erase( iterator pos )`: removes the object at position `pos`. The method returns an iterator to the element that follows `pos` before the call. This operation *invalidates* `pos`, since the item it pointed to was removed from the list.
- `iterator erase( iterator first, iterator last )`: removes elements in the range `[first; last)`. The entire list may be erased by calling `a.erase(a.begin(), a.end());`
- `template <typename InItr>`
  `void assign( InItr first, InItr last )`: replaces the contents of the list with copies of the elements in the range `[first; last)`.
- `void assign( std::initializer_list<T> ilist )`: replaces the contents of the list with the elements from the initializer list `ilist`.
  We may call, for instance, `myList.assign( {1, 2, 3, 4} )`, to replace the elements of the list with the elements 1, 2, 3, and 4, assuming that `myList` is a list of `int`.

For the particular case of List ADT being implemented with deque, you should notice that insertion operations may (1) cause reallocation if the new `size()` is greater than the `capacity()`, and (2) make all previous iterators and references invalid.

4 Implementation

Your mission, should you choose to accept it, is to implement a class called `deque` that follows the List ADT design described previously. Internally, the deque keeps an *dynamic array* of pointers to blocks (i.e fixed-size arrays) of data. Let us call this dynamic array a **map of blocks** or **mob**.

During the lifetime of a deque, new blocks are allocated dynamically inside the *mob*, depending on storage needs. Also, the *mob* itself might be expanded, if an insertion operation is requested and all its pointers are already pointing to a block fully occupied. In this case the we must **double** its current storage capacity, and copy all the pointers from the old *mob* to the new one.

Because we are only copying pointers, the blocks that actually hold the elements are the same. This means we do not need to copy the current data element-wise, but rather block-wise. Therefore the number of copy operations are greatly reduced, if compared to a regular list, say a ‘std::vector‘, with the same amount of elements. See Figure 2 for an abstract example of this idea. Note that this doubling-memory-and-copying-data action should go completely unnoticed to the client code.

Try to use `std::shared_ptr` so that the release of resources is automatically taken care of. This behavior is called *Resource Acquisition Is Initialization* (RAII), which basically binds the life

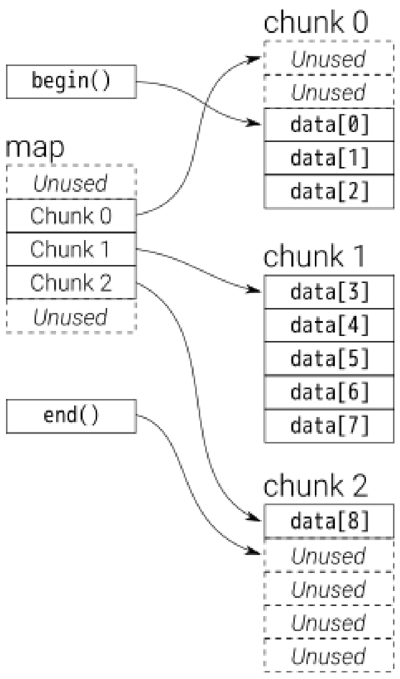

Figura 2: Internal representation of a deque.

cycle of a resources (in the `deque`'s case, the dynamic array) to the lifetime of an object (in the case, the `deque` itself).

In practical terms, this means that when the life cycle of a `deque` object ends, the internal dynamic array is automatically deleted, without the need to put the `delete []` command in the destructor `~deque()`.

You should declare a nested `const_iterator` class (so it has access to the dynamic array) that basically is friend of the `deque` class and holds a private regular pointer. The same goes to the class `iterator`, which means it is also a nested class.

All classes may be declared in the same file, named `deque.h`.

4.1 Unit Test Code

I **strongly** recommend that you make use of the *unit tests* provided with this project. The unit tests are an automatic way of testing each of the methods of your class under various situations. The application of the test suits during the development phase helps you pinpoint which methods are working as expected or not at all. Besides, if your class pass all tests it means you are in the right track to get full credits.

Note, however, that the unit tests do not ensure that your code is 100% correct. It only shows that the class works for some basic circumstances. Other aspects, such as memory leaks, or unforeseen bugs may not be caught by the tests alone.

5 Assignment Evaluation

You should hand in a complete program, without compiling errors, tested and fully documented. The assignment will be credit according to the following criteria:-

1. Correct implementation of `deque`:
    - (a) Special members (35 credits);
        - i. Regular constructor $\times 3$ (12 credits);
        - ii. Destructor (3 credits);
        - iii. Copy constructor (4 credits);
        - iv. Constructor from range (4 credits);
        - v. Constructor from initialize list (4 credits);
        - vi. Assignment operator $\times 2$ (8 credits);
    - (b) Iterator retrieval methods (8 credits);:
        - i. `begin()` (2 credits);
        - ii. `end()` (2 credits);
        - iii. `cbegin()` (2 credits);
        - iv. `cend()` (2 credits);
    - (c) Capacity methods (3 credits);
        - i. `empty()` (1 credits);
        - ii. `size()` (1 credits);
        - iii. `capacity()` (1 credits);
    - (d) Modifiers methods (40 credits);
        - i. `clear()` (4 credits);
        - ii. `push_front()` (4 credits);
        - iii. `pop_front()` (4 credits);
        - iv. `push_back()` (4 credits);
        - v. `pop_back()` (4 credits);
        - vi. `insert()` $\times 4$ (20 credits);
        - vii. `resize()` (4 credits);
        - viii. `shrink_to_fit()` (3 credits);
        - ix. `assign()` $\times 2$ (6 credits);
        - x. `erase()` $\times 2$ (8 credits);
    - (e) Element access methods (18 credits); and,
        - i. `front()` (1 credits);
        - ii. `back()` (1 credits);
        - iii. `operator[]()` $\times 2$ (8 credits);
        - iv. `at()` $\times 2$ (8 credits);
    - (f) Operators (2 credits).
        - i. `operator==()` (1 credits);

   ii. `operator!=()` (1 credits);

2. Iterator classes
   (a) Special members (10 credits);
      i. Regular constructor (4 credits);
      ii. Copy constructor (4 credits);
      iii. Assignment operator (2 credits);
   (b) Navigation methods (37 credits);:
      i. increment operator `++it` and `it++` (4 credits);
      ii. decrement operator `--it` and `it--` (4 credits);
      iii. forward jump operator `it + n` or `n + it` (6 credits);
      iv. backward jump operator `it - n` (3 credits);
      v. forward self jump operator `it += n` (4 credits);
      vi. backward self jump operator `it -= n` (4 credits);
      vii. difference between iterators `it1-it2` (4 credits);
      viii. dereference operator `*it` (2 credits);
      ix. equality/difference operators `it1==it2` and `it1!=it2` (2 credits);
      x. greater/less than operators `it1<it2` and `it1>it2` (2 credits);
      xi. greater/less than operators `it1<=it2` and `it1>=it2` (2 credits);

The following situations may *take credits out* of your assignment, if they happen during the evaluation process:-

- Project does not support the unit tests provided (up to −20 credits)
- Compiling and/or run time errors (up to −20 credits)
- Missing code documentation in Doxygen style (up to −10 credits)
- Memory leak (up to −10 credits)
- Missing `author.md` file (up to −20 credits).

The `author.md` file (Markdown file format recommended here) should contain a brief description of the project, and how to run it. It also should describe possible errors, limitations, or issues found. Do not forget to include the author(s) name(s)!

Good Programming Practices

During the development process of your assignment, it is strongly recommend to use the following tools:-

- Doxygen: professional code documentation;
- Git: version control system;
- Valgrind: tracks memory leaks, among other features;
- gdb: debugging tool, and;
- Makefile: helps building and managing your programming projects.

Try to organize you code in several folders, such as `src` (for `.cpp` files), `include` (for header files `.h`, and `.inl`), `bin` (for `.o` and executable files) and `data` (for storing input files).

6 Authorship and Collaboration Policy

This is a pair assignment. You may work in a pair or alone. If you work as a pair, comment both members' names atop every code file, and try to balance evenly the workload. Only one of you should submit the program via Sigaa.

Any team may be called for an interview. The purpose of the interview is twofold: to confirm the authorship of the assignment and to identify the workload assign to each member. During the interview, any team member should be capable of explaining any piece of code, even if he or she has not written that particular piece of code. After the interview, the assignment's credits may be re-distributed to better reflect the true contribution of each team member.

The cooperation among students is strongly encouraged. It is accepted the open discussion of ideas or development strategies. Notice, however, that this type of interaction should not be understood as a free permission to copy and use somebody else's code. This is may be interpreted as plagiarism.

Any two (or more) programs deemed as plagiarism will automatically receive **zero** credits, regardless of the real authorship of the programs involved in the case. If your project uses a (small) piece of code from someone else's, please provide proper acknowledgment in the `author.md` file.

7 Work Submission

You may submit your work in two possible ways: via GitHub Classroom (GHC), or, via Sigaa submission task. In case you decide to send your work via GHC you **must** also send a text file via Sigaa submission task with the github link to your repository. In case you choose to send your work via Sigaa only, send a zip file containing all the code necessary to compile and run the project.

I any of these two ways, remember to remove all the executable files (i.e. the `build` folder) from your project before handing in your work. Only one team member should submit a single zip file containing the entire project. This should be done only via the proper link in the Sigaa's virtual class.

◀ The End ▶