# Doubly-Linked-List in Rust

## Documentation

# Introduction

Doubly Linked List is the most primitive data structure after the Singly Linked List. Although you can easily find resources on the internet about the implementation of Doubly Linked List in C++, Java, Python, etc. it’s very hard to find code of this data structure, implemented in Rust programming language. But why is it so hard to implement Doubly Linked List in Rust? What makes Rust so different from other vastly used programming languages? Why should we use Rust at all? What makes it hard to build Doubly Linked in Rust? After reading this documentation we hope that you will have answers to these questions and also you will be able to see the difference between Rust and other widely exploited programming languages as you follow us into the process of thinking and implementation.

Throughout our documentation, we will explain important concepts of Rust, that make it unique, fast, and performant. We will examine how Rust manages memory to understand how should we use information stored in memory. After all, we will see what are the restrictions referring to using pointers and references. Our goal is to explain every feature of the language, which will be used later, in easily understandable language.

During the planning of our model of data structure, we will analyze problems, facing off Rust restrictions and memory management. After analysis, solutions will be introduced, explained, and estimated. As we will be able to use memory efficiently, we will try to make the time complexity of the data structure as effective as possible.

In the process of the implementation, code will be well structured, easily readable, and understandable. Methods in the data structure will be split up as much as possible according to their functionality, using that way of arrangement we will get rid of redundant code and our scripts will be much easier to read, imagine and understand. Names of the structure data types and methods will also be as self-descriptive as possible. Test examples will be written and provided to check the efficiency of our code at every stage of progress.

# Brief Review About Rust

Rust has its distinctive features, that make it safe and fast. In order to develop our general understanding of the programming language, we need to learn how ownership, references, and lifetimes work. We are assuming that you know some basic syntax of Rust and you can understand C++, Let’s dive in!

## Pointers and References

To take the initial step, we need to know how pointers work in Rust. It’s important to understand this topic because without pointers, it’s nearly impossible to create a data structure that will use memory efficiently.

In Rust, we have several types of smart pointers. But at first, let’s explain what is a pointer in general, and what are those smart pointers? So, a pointer is just a variable that stores the memory address of another variable. At first, we should see what address of variable looks like.

Let’s see C++ example:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. cout << &x << endl;
6. }

Output:

0x7ffd9a45e5e4

As we can see if we print out variable, with ‘&’ sign we will get a **hexadecimal number**, which actually is an address in our memory. In programming that **hexadecimal address** is called **reference** of a variable.

We can also see what is stored in the reference using ‘\*’ sign like that:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. cout << \*&x << endl;
6. }

Output:

10

The process of seeing what is stored of reference is called **dereferencing**.

It would be pity, if we couldn’t save reference of variable somewhere for future use, it’s also little bit annoying to use ‘&’ sign when you want to get the reference of some variable. That’s why we have pointers in C++ and declaration and use of pointer looks like this:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. int\* x\_ptr = &x;
6. cout << x\_ptr << endl;
7. }

Output:

0x7ffc137306cc

 We can also dereference our pointer like that:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. int\* x\_ptr = &x;
6. cout << \*x\_ptr << endl;
7. }

 Output:

10

 As you can see ‘\*’ sign is also used for declaration of a pointer, so please don’t confuse, when writing <Type> \* <Variable Name> like int\* x\_ptr = &x it means that we are using ‘\*’ sign for declaration of the pointer. But When we write \*<Variable Name> like: \*x\_ptr it means **dereferencing**.

Please also note that, it doesn’t matter how will you write:

int\* x\_ptr = &x;

or

int \*x\_ptr = &x;

but the last example is more preferable (although we use both syntax), since when you want to declare multiple pointer in one line you have to write like that:

int \*x\_ptr = &x, \*y\_ptr = &y;

Pointers also can be declared with new keyword in order to allocate memory for new pointer variable. After allocation we can store reference to it. Let’s Check:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. int\* x\_ptr = new int;
6. x\_ptr = &x;
7. cout << \*x\_ptr << endl;
8. }

Output:

10

Memory allocation with a “new” keyword is widely used. For example, we use it to create a new pointer for a new element or node of some data-structure.

Sometimes after usage of pointers, they become redundant. Also at some point in our code, we don’t need some variables and we want to free out some memory. Therefore C++ provides us with a delete keyword. So, if we want to remove our variable from memory our code will look like this:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. // do some stuff with x
6. delete &x;
7. cout << x << endl;
8. }

Output:

munmap\_chunk(): invalid pointer

Command terminated by signal 6

In output we would definitely get error, because memory was emptied. We can also do the same with pointers.

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. int \*x\_ptr = new int;
6. x\_ptr= &x;
7. delete x\_ptr;
8. cout<<x\_ptr<<endl;
9. }

 Output:

munmap\_chunk(): invalid pointer

Command terminated by signal 6

Please note, we also can’t allocate new space for deleted pointer:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x = 10;
5. int \*x\_ptr = new int;
6. x\_ptr = &x;
7. delete x\_ptr;
8. x\_ptr = new int;
9. int y = 5;
10. x\_ptr = &y;
11. cout<< x\_ptr <<endl;
12. }

Output:

free(): invalid pointer

Command terminated by signal 6

The kind of pointers, that we have just explained, is also called a raw pointer. When we use raw pointers, we are responsible to delete them whenever it’s necessary.

Now let’s consider the following example of using raw pointers:

1. #include <iostream>
2. using namespace std;
4. void mem\_leak\_func(){
5. int \*ptr = new int;
6. //do some stuff with pointer
7. //return without deallocation
8. return;
9. }
11. int main() {
12. mem\_leak\_func();
13. cout<<"successfully executed!!!";
14. }

Output:

successfully executed!!!

We allocated space in memory for the pointer, which was not deallocated but became inaccessible for main() function as leaving mem\_leak\_func() function scope.

So, after the execution of the mem\_leak\_func() function, we have allocated space in memory that can’t be reached. This phenomenon is called **Memory Leak**, execution following function many times will fill up whole memory.

1. #include <iostream>
2. using namespace std;
4. void mem\_leak\_func(){
5. int\* ptr = new int;
6. //do some stuff with pointer
7. //return without deallocation
8. return;
9. }
11. int main() {
12. for(int i=0;i<25000000;i++){
13. mem\_leak\_func();
14. }
15. cout<<"successfully executed!!!";
16. }

Output:

terminate called after throwing an instance of 'std::bad\_alloc'

what(): std::bad\_alloc

We got bad allocation runtime error because memory is full and it’s impossible to allocate new space. So, if we ever do not deallocate used space we may get a runtime error that is hard to find and it’s also very easy to forget deallocation when code size becomes bigger.

In order to avoid **memory leaks**, we use **Smart Pointers**. Smart Pointers deallocate space in memory when it’s necessary. In C++ one of the Smart Pointers is called unique\_ptr, which points to a single value. Check our code using Smart Pointer:

1. #include <bits/stdc++.h>
2. using namespace std;
4. void mem\_leak\_func(){
5. unique\_ptr<int> valuePtr(new int);
6. //do some stuff with pointer
7. //return without deallocation
8. return;
9. }
11. int main() {
12. for(int i=0;i<25000000;i++){
13. mem\_leak\_func();
14. }
15. cout<<"successfully executed!!!";
16. }

Output:

successfully executed!!!

We can also get dangling pointers if we execute code:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int \*x\_ptr = new int(55);
5. int \*y\_ptr = x\_ptr;
6. //print out memory addresses that are saved in pointers
7. cout<<"before x\_ptr deletion"<<endl;
8. cout<<"x\_ptr : "<<x\_ptr<<endl;
9. cout<<"y\_ptr : "<<y\_ptr<<endl;
10. //print out line break
11. cout<<"-----\n";
12. delete x\_ptr;
13. x\_ptr = nullptr;
14. // address of x\_ptr is freed but y\_ptr still points to it
15. cout<<"after x\_ptr deletion"<<endl;
16. cout<<"x\_ptr : "<<x\_ptr<<endl;
17. cout<<"y\_ptr : "<<y\_ptr<<endl;
18. }

Output:

before x\_ptr deletion

x\_ptr : 0x556e9aeafeb0

y\_ptr : 0x556e9aeafeb0

-----

after x\_ptr deletion

x\_ptr : 0

y\_ptr : 0x556e9aeafeb0

As we can see Smart Pointers have advantages over Raw Pointers. Maybe one of the disadvantages of Smart Pointers is that it takes more time to instantiate Smart Pointer compared to Raw Pointer, but the time difference is **so small** that, it doesn’t worth using Raw Pointers if we don’t work for a special kind of project of course.

Now, as we can see the difference between Smart and Raw Pointers. We can speak about Smart Pointers in Rust.

Rust also has Raw and Smart Pointers, but raw pointers are only used in Unsafe Rust because Memory Leak is a serious security issue. In our project we will not use any Pointer that is unsafe to use, that’s why we will take into perspective only Smart Pointers.

There are several kinds of Smart Pointers in Rust:

**Box Pointers** Box<T> **:** One variable can’t have two or more **Box Pointers**. Variables stored in this kind of Smart Pointer **can be** dereferenced and edited (mutated).

**Reference Counting Pointers** Rc<T> : Compared to Box Pointers, a single variable can have Multiple **Reference Counting Pointers**, but variables inside this kind of Smart Pointer, **can’t be** dereferenced and edited (mutated).

**Reference Cell Pointers** RefCell<T> : This type enforces the borrowing rules at runtime instead of compile-time. This kind of pointers can be used with weak pointers as an escape from safety rules, therefore we won’t use them because our data structure has to be safe and efficient.

In our project we will heavily use Box Pointers and references, so you don’t need to worry about another kind of pointers.

Note that: **In Rust, there is no NULL pointer** instead of that it’s used Option::None which is Enum type.

## Syntax of References in Rust

Rust treats references differently than other programming languages, we will examine and compare C++ and Rust examples in order to understand conceptual differences and similarities between Rust and C-like languages approach to the references.

If we want to declare reference variable in C++, we should do like that:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x=10;
5. int &x\_ref = x;
6. cout << x\_ref << endl;
7. }

Output:

10

 In C++, it’s more look like an alias for some variable, every change applied to x\_ref variable will also be applied to x variable.

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int x=10;
5. int &x\_ref = x;
6. x\_ref++;
7. cout << x\_ref << endl;
8. }

 Output:

11

 But why do we need an alias for the variable if we can access it with the original name? We need reference variables to do not copy them when we passing an argument to the function. So, let’s look at that:

1. #include <iostream>
2. using namespace std;
3. void increase\_variable(int var){ // copy of x value saved in var
4. var++; // copy of x value increased
5. }
6. int main() {
7. int x=10;
8. increase\_variable(x);
9. cout << x << endl;
10. }

 Output:

10

x variable have not increased, because increase\_variable function created **copy of x value** and x was not increased. But if we write like that:

1. #include <iostream>
2. using namespace std;
3. void increase\_variable(int &var){ // reference of x copied in saved in var
4. var++; // var increased, so x increased also, as var is the reference of x
5. }
6. int main() {
7. int x=10;
8. increase\_variable(x);
9. cout << x << endl;
10. }

 Output:

11

As you can see only **reference of x was copied** instead of x, so we avoided copying of the variable, the only reference was copied and x increased as **the reference of x** was increased. Maybe it still seems a little bit confusing but whenever you have a big data structure like vector, for instance, you should pass the function the reference of vector variable to avoid copying the whole data of vector.

Now let’s see how that works in Rust, if we want to declare reference for a value it will look like this:

1. fn main() {
2. let mut x : i32 = 10;
3. let x\_ref : &mut i32 = &mut x;
4. \*x\_ref+=1;
5. println!("{}",\*x\_ref);
6. }

Output:

11

 As you can see if you want to declare variable in Rust you have to use following syntax let [<mutability>] <variable name> : <type> = <value> (i32 stands for 32-bit integer). Note that, by default in rust variables are not mutable so we have to use mut keyword if we want to declare a mutable variable. Syntax of this code can be sugarized since we can do not specify a type, if the compiler can infer it independently.

let mut x : i32 = 10; can be sugarized as let mut x = 10;. we will not use sugar syntax, intentionally for better comprehension.

Now let’s discuss the third line of the code shown above, and compare it to the C++ syntax:

In C++:

int &x\_ref = x;

In rust:

let x\_ref : &mut i32 = &mut x;

In C++ we use ‘&’ sign on the left side of the equal sign (‘=’), but In Rust ‘&’ sign appears on the right side. We also have to use mut keyword with ‘&’ sign if we are storing reference of the mutable variable to the x\_ref.

Now let’s see what is happening on the fourth line.

In C++:

x\_ref++;

In Rust:

\*x\_ref+=1;

You have to explicitly dereference the reference variable in Rust in order to change it, but it is not happening in C++.

If we want to increase our variable using function our code will be like this:

1. fn increase\_variable(var : &mut i32){
2. \*var+=1;
3. }
4. fn main() {
5. let mut x : i32 = 10;
6. increase\_variable(&mut x);
7. println!("{}",x);
8. }

Output:

11

Now we know the syntax of rust references, let’s move on to the next concept called Ownership!!!

## Ownership

Ownership is one of the core concepts of the rust that distinguishes it from other languages. Several rules are referring to ownership:

* Each value in Rust has a variable that’s called its *owner*.
* There can only be one owner at a time.
* When the owner goes out of scope, the value will be dropped.

These rules make our code safer but it’s a little bit hard to get used to it. So, let see everything in detail. Instead of copying, in Rust values are moved by default because “There can only be one owner at a time”. This feature saves us from dangling-pointer-error. But there are primitive types that can be copied.For example Integer types (i8,u32,i128,usize), Float Point Types (f32,f64), Boolean Types (true, false) and Character Types (‘A’, ‘😻’ , ’x’). But any other types are regarded as Compound types Like Strings, Vectors, mutable references (&mut), etc.

If we follow Rust rules we will avoid memory leaks! Let’s consider the following code in C++:

1. #include <iostream>
2. using namespace std;
3. int main() {
4. int \*x\_ptr = new int(10);
5. int \*y\_ptr = new int(20);
6. //initial values stored in pointers
8. cout<<"before reassignment of y\_ptr"<<endl;
9. cout<< x\_ptr <<endl;
10. cout<< y\_ptr <<endl;
12. y\_ptr = x\_ptr;//y\_ptr reassigned
14. cout<<"before reassignment of y\_ptr"<<endl;
15. cout<< x\_ptr <<endl;
16. cout<< y\_ptr <<endl;
18. }

Output:

before reassignment of y\_ptr

0x55691a835eb0

0x55691a835ef1

before reassignment of y\_ptr

0x55691a835eb0

0x55691a835eb0

As you can see after reassignment of y\_ptr we have no access to 0x55691a835ef1 address, and we are getting memory leak.

Now let’s do the similar thing with **Box** Type **Smart Pointer** to **Compound Type** like &mut:

1. #[allow(unused)]
2. fn main() {
3. let mut x\_ptr : Box<&mut i32> = Box::new(&mut 10);
4. let mut y\_ptr : Box<&mut i32> = Box::new(&mut 20);
5. x\_ptr = y\_ptr;
6. }

Output:

1. error[E0716]: temporary value dropped while borrowed
2. --> jdoodle.rs:4:51
3. |
4. 4 | let mut y\_ptr : Box<&mut i32> = Box::new(&mut 20);
5. | ^^ - temporary value is freed at the end of this statement
6. | |
7. | creates a temporary which is freed while still in use
8. 5 | x\_ptr = y\_ptr;
9. | ----- borrow later used here
10. |
11. = note: consider using a `let` binding to create a longer lived value
13. error: aborting due to previous error
15. For more information about this error, try `rustc --explain E0716`.

Rust do not allow us to leave temporary value inaccessible. Thanks to Rust restrictions we can avoid memory leaks!!! We cannot use variables of moved values, for instance:

1. #[allow(unused)]
2. fn main() {
3. let mut x\_ptr : Box<i32> = Box::new(10);
4. let mut y\_ptr : Box<i32> = Box::new(20);
5. x\_ptr = y\_ptr;
6. println!("{} {}",\*x\_ptr,\*y\_ptr);
7. }

Output:

1. error[E0382]: borrow of moved value: `y\_ptr`
2. --> jdoodle.rs:6:29
3. |
4. 4 | let mut y\_ptr : Box<i32> = Box::new(20);
5. | --------- move occurs because `y\_ptr` has type `std::boxed::Box<i32>`, which does not implement the `Copy` trait
6. 5 | x\_ptr = y\_ptr;
7. | ----- value moved here
8. 6 | println!("{} {}",\*x\_ptr,\*y\_ptr);
9. | ^^^^^^ value borrowed here after move
11. error: aborting due to previous error
13. For more information about this error, try `rustc --explain E0382`.

Yeah, we were dereferencing moved value, so we got error here. But if we want to copy Compound Types we can use clone() method:

1. #[allow(unused)]
2. fn main() {
3. let mut x\_ptr : Box<i32> = Box::new(10);
4. let mut y\_ptr : Box<i32> = Box::new(20);
5. x\_ptr = y\_ptr.clone();
6. println!("{} {}",\*x\_ptr,\*y\_ptr);
7. }

Output:

20 20

 Rust forces us to use the clone() method to copy!

Ownership concept and moving semantics are steadily connected with function scopes, because as we pass variable to the function, compound values are moved to function scope.

Let’s create String and try to print that two times using function:

1. fn print\_string(s: String){ //word value moved here in s
2. println!("{}",s); // move value printed out
3. }//moved value stored in s dropped as it left the function scope
4. fn main() {
5. let word : String = String::from("hello!");
6. print\_string(word);
7. print\_string(word);
8. }

Output:

1. error[E0382]: use of moved value: `word`
2. --> jdoodle.rs:7:18
3. |
4. 5 | let word : String = String::from("hello!");
5. | ---- move occurs because `word` has type `std::string::String`, which does not implement the `Copy` trait
6. 6 | print\_string(word);
7. | ---- value moved here
8. 7 | print\_string(word);
9. | ^^^^ value used here after move
11. error: aborting due to previous error
13. For more information about this error, try `rustc --explain E0382`.

Code didn’t compile successfully because, our String type variable, stored in word, moved into the function scope and it dropped after the last line of print\_string() function executed, so value stored in the word didn’t get back it’s owned variable and became empty. As it became empty, we were not allowed to use it again, therefore the compiler gave us an error about the usage of the valueless variable.

In order to avoid this error, we just need to pass the reference of our variable to the function. The process of passing variable reference is called **borrowing**.

1. fn print\_string(s: &String){ //reference to word variable copied into s
2. println!("{}",\*s); //value stored in reference printed out
3. }//reference of word variable destroyed but not the word value itself.
4. fn main() {
5. let word : String = String::from("hello!");
6. //borrow word variable to the function
7. print\_string(&word);//passed only reference not value
8. print\_string(&word);//passed only reference not value
9. }

Output:

hello!

hello!

We reached our goal (without cloning of value) using the **borrow** concept.

## Lifetimes

In Rust every variable has its own lifetime, to understand Lifetimes let’s start from the easiest example, like other programming languages, Rust also has code blocks with its scope. Let’s write code using code blocks.

*fn* do\_some\_stuff() {  
 *let* x: &*i32* = &10;  
 {  
 *let* z: &*i32* = x;  
 *let* y: &*i32* = &15;  
 print!("{} + {} = {}", \*z, \*y, \*z + \*y);  
 }  
}  
  
*fn* main() {  
 do\_some\_stuff();  
}

Output:

10 + 15 = 25

Everything works fine since rust can infer the lifetime of each variable, but what we see is sugarized syntax. So, we can get the same result with lifetime variables, let’s see what our code will look like if we add lifetime variable to the x variable.

*fn* do\_some\_stuff<'a>() {  
 *let* x: &'a *i32* = &10;  
 {  
 *let* z: &'a *i32* = x;  
 *let* y: &*i32* = &15;  
 print!("{} + {} = {}", \*z, \*y, \*z + \*y);  
 }  
}  
  
*fn* main() {  
 do\_some\_stuff();  
}

Output:

10 + 15 = 25

It’s obvious that the lifetime of x is longer than the lifetime of y and z, but when we write in the **Type Field** of z: &'a *i32* it means, that we are storing in z value of *i32* variable, which has a longer lifetime than z.

We can also specify the lifetime of y, to show that y and z are taking values, that have different lifetimes:

*fn* do\_some\_stuff<'a, 'b>() {  
 *let* x: &'a *i32* = &10;  
 {  
 *let* z: &'a *i32* = x;  
 *let* y: &'b *i32* = &15;  
 print!("{} + {} = {}", \*z, \*y, \*z + \*y);  
 }  
}  
  
*fn* main() {  
 do\_some\_stuff();  
}

Output:

10 + 15 = 25

We can make our code little bit complicated, using more lifetimes:

*fn* do\_some\_stuff<'a, 'b, 'c>(var : &'c *i32*) {  
 *let* x: &'a *i32* = &10;  
 {  
 *let* z: &'a *i32* = x;  
 *let* y: &'b *i32* = &15;  
 print!("{} + {} + {} = {}", \*z, \*y, \*var, \*z + \*y + \*var);  
 }  
}  
  
*fn* main() {  
 *let* variable: &*i32* = &20;  
 do\_some\_stuff(variable);  
}

Output:

10 + 15 + 20 = 45

Please note, we **cannot** specify lifetime parameters in the main() function:

*fn* do\_some\_stuff<'a, 'b, 'c>(var: &'c *i32*) {  
 *let* x: &'a *i32* = &10;  
 {  
 *let* z: &'a *i32* = x;  
 *let* y: &'b *i32* = &15;  
 print!("{} + {} + {} = {}", \*z, \*y, \*var, \*z + \*y + \*var);  
 }  
}  
  
*fn* main<'c>() {  
 *let* variable: &'c *i32* = &20;  
 do\_some\_stuff(variable);  
}

Output:

error[E0131]: `main` function is not allowed to have generic parameters  
--> src\main.rs:10:8  
|  
10 | *fn* main<'c>() {  
 | ^^^^ `main` cannot have generic parameters  
  
 error: aborting due to previous error  
  
 For more information about this error, try `rustc --explain E0131`.

We can create *struct*, which will take references of different lifetimes. At first, let’s see what will happen if we don’t specify lifetime parameter for *struct*.

*struct* Coordinate {  
 x: &*i32*}  
  
*fn* main() {  
 *let* s: Coordinate;  
 {  
 *let* position: *i32* = 10;  
 s = Coordinate {  
 x: &position  
 };  
 }// position value drops here  
 print!("{}", \*s.x); //position value is already dropped  
}

Output:

error[E0106]: missing lifetime specifier  
--> src\main.rs:2:8  
|  
2 | x: &i32  
| ^ expected named lifetime parameter  
|  
help: consider introducing a named lifetime parameter  
|  
1 | *struct* Coordinate<'a> {  
2 | x: &'a i32  
|  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0106`.

Rust forces us to use lifetime specifiers for structure keys because if we don’t specify a lifetime, the value of some key might be dropped. If the structure key has dropped value it means that in the structure we are keeping dangling reference. In order to do not use *struct* instance after its value is dropped, we have to explicitly declare lifetime parameters for *struct* s.

*struct* Coordinate<'a> {  
 x: &'a *i32*}  
  
*fn* main() {  
 *let* s: Coordinate;  
 {  
 *let* position: *i32* = 10;  
 s = Coordinate {  
 x: &position  
 };  
 }  
 print!("{}", \*s.x);  
}

Output:

error[E0597]: `position` does not live long enough  
--> src\main.rs:10:16  
|  
10 | x: &position  
| ^^^^^^^^^ borrowed value does not live long enough  
11 | };  
12 | }  
| - `position` dropped here whilestill borrowed  
13 | print!("{}", \*s.x);  
| ---- borrow later used here  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0597`.

We got an error because the value stored in x is already dropped and it means that we are using *struct* variable at an improper place. We have to print out the key of the structure before the value of x will be dropped.

So, proper code looks like this:

*struct* Coordinate<'a> {  
 x: &'a *i32*}  
  
*fn* main() {  
 *let* s: Coordinate;  
 {  
 *let* position: *i32* = 10;  
 s = Coordinate {  
 x: &position  
 };  
 print!("{}", \*s.x);  
 }  
}

Output:

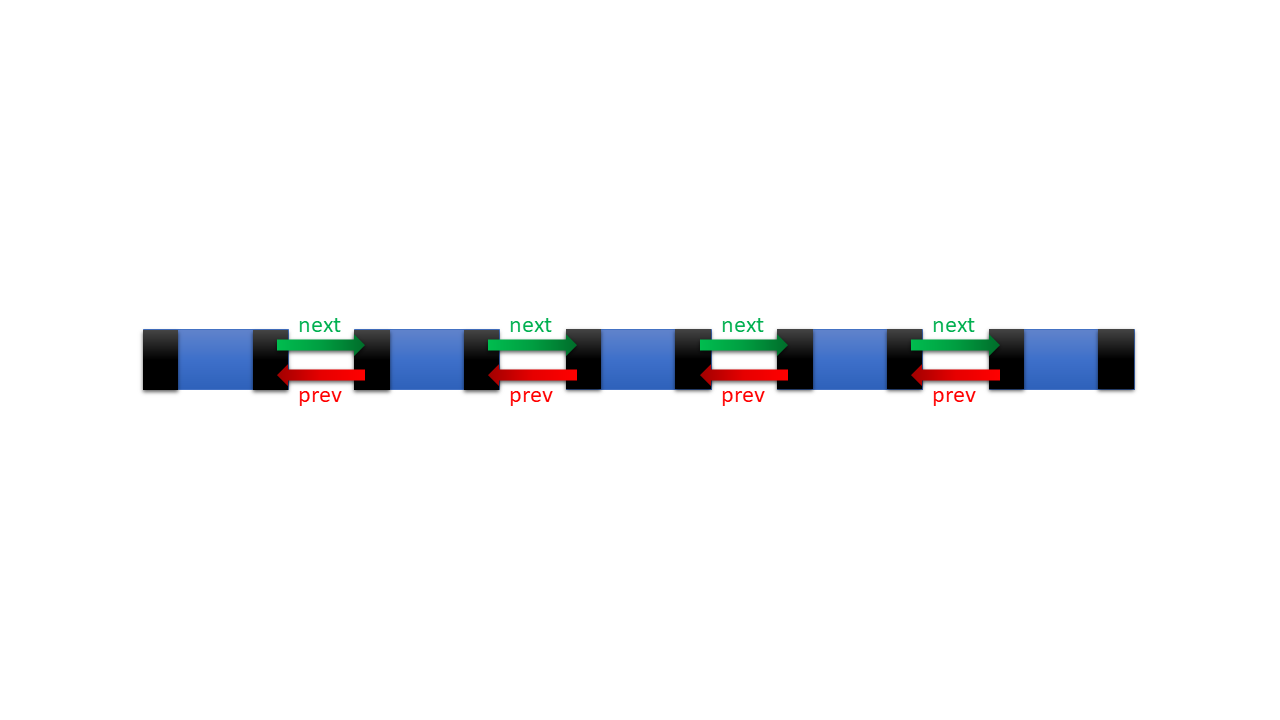
10

In summary, lifetimes help us to make our code safer and more comprehensive. Now we have an understanding of Rust’s core concepts. Using these features of Rust, we will create a safe and fast data structure.

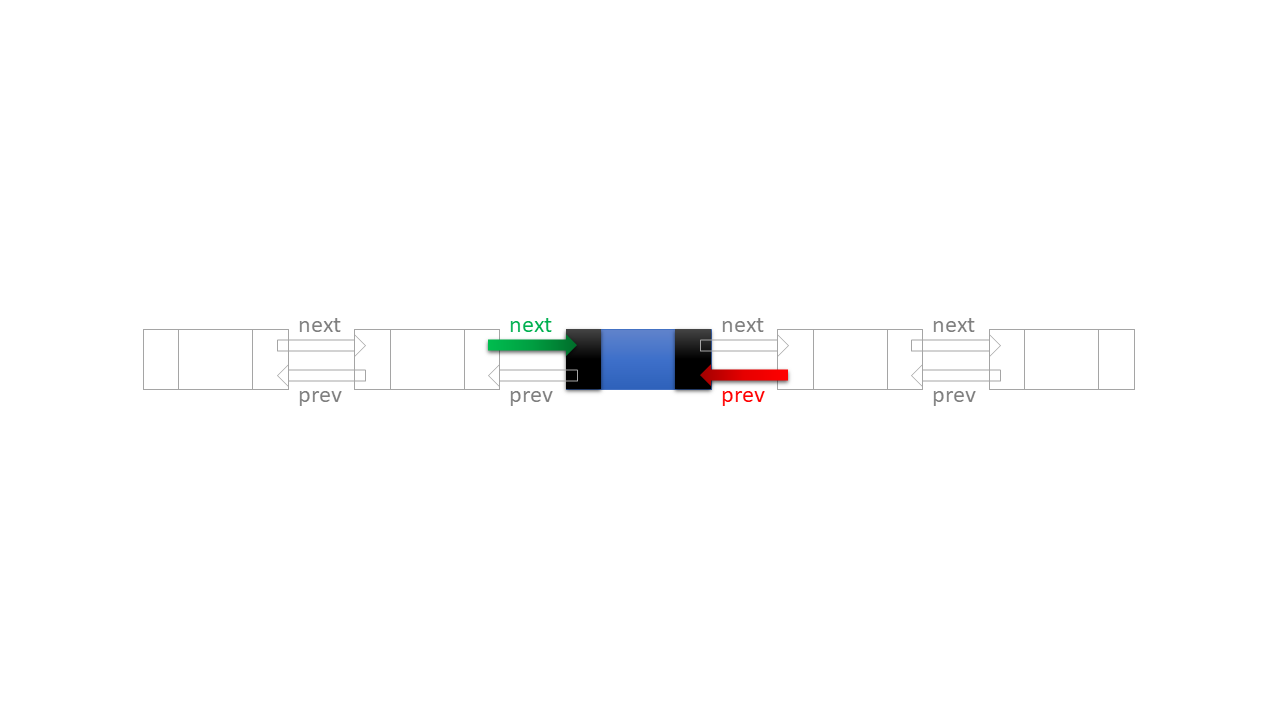
# Constructing Doubly Linked List in Rust

In this part we will find out the way of constructing the Doubly-Linked-List data structure in Rust, using safe and effective approaches.

Before the explanation of the solution, let’s see how does Doubly-Linked-List looks like and what are the challenges we are facing.



Each node has two pointers called the next and previous. And that leads us to the problem: some of the nodes pointed by two pointers.



In Rust we cannot have two mutable pointers, pointing to the same object. But why do we need a pointer in a node named next and previous? We need them to navigate left and right in Doubly Linked List whenever we want.

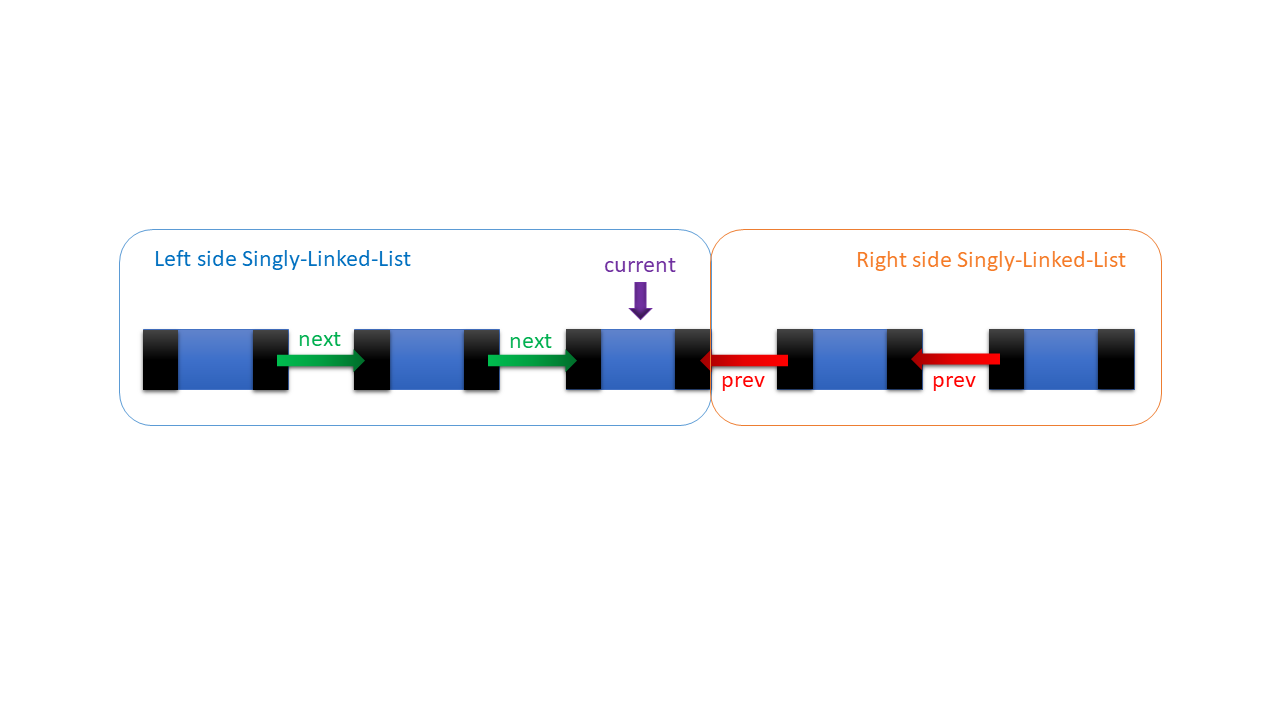
Two pointers, pointing to the same object, create a safety issue. Because if reference of the first pointer will be dropped, then in the second dangling reference will be left.

Given that, Doubly-Linked-List seems unsafe data structure. But is there any way to make that structure safe and keep the functionality and benefits (traversing either left or right at any stage) of this data structure?

Fortunately, the answer to that question is positive. It sounds a little bit weird, but to keep functionality and benefits of the structure we don’t need to have two pointers for each node.

We can keep the data structure 100% safe if we split our list into the left side Singly-Linked-List and right side Singly-Linked-List in relation to the current node.

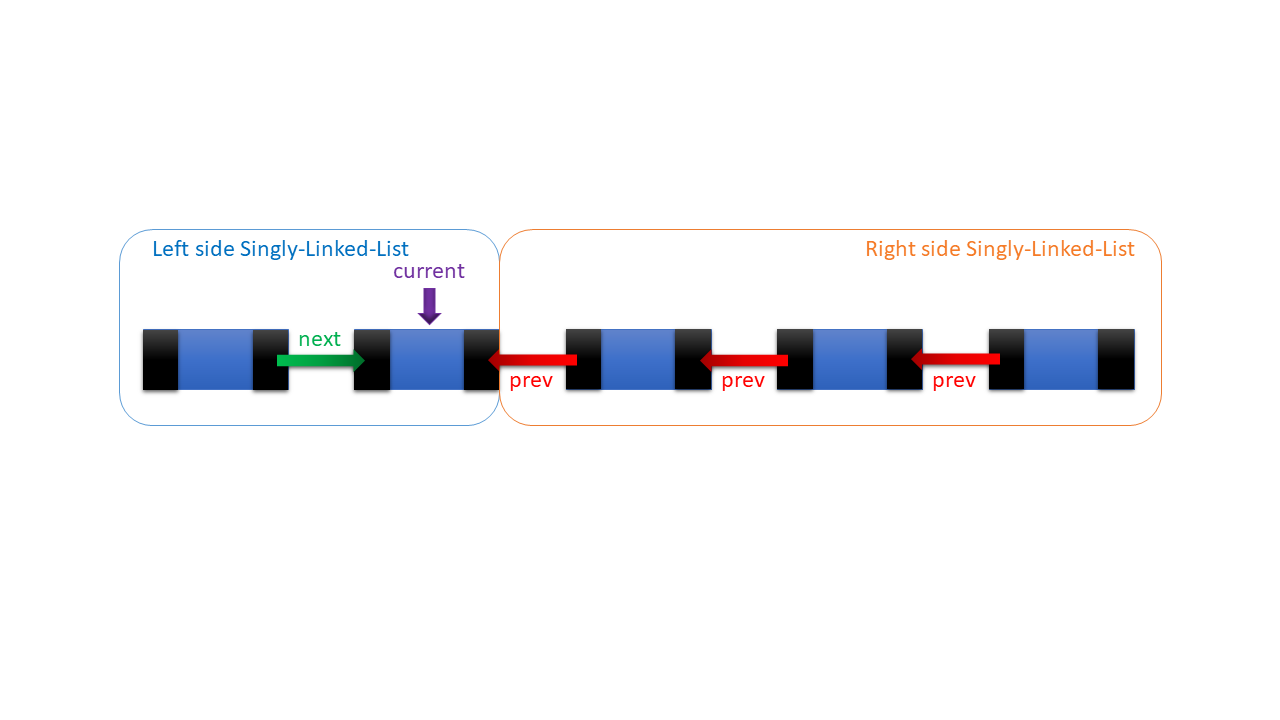
Visual of our Doubly-Linked-List should be something like that:



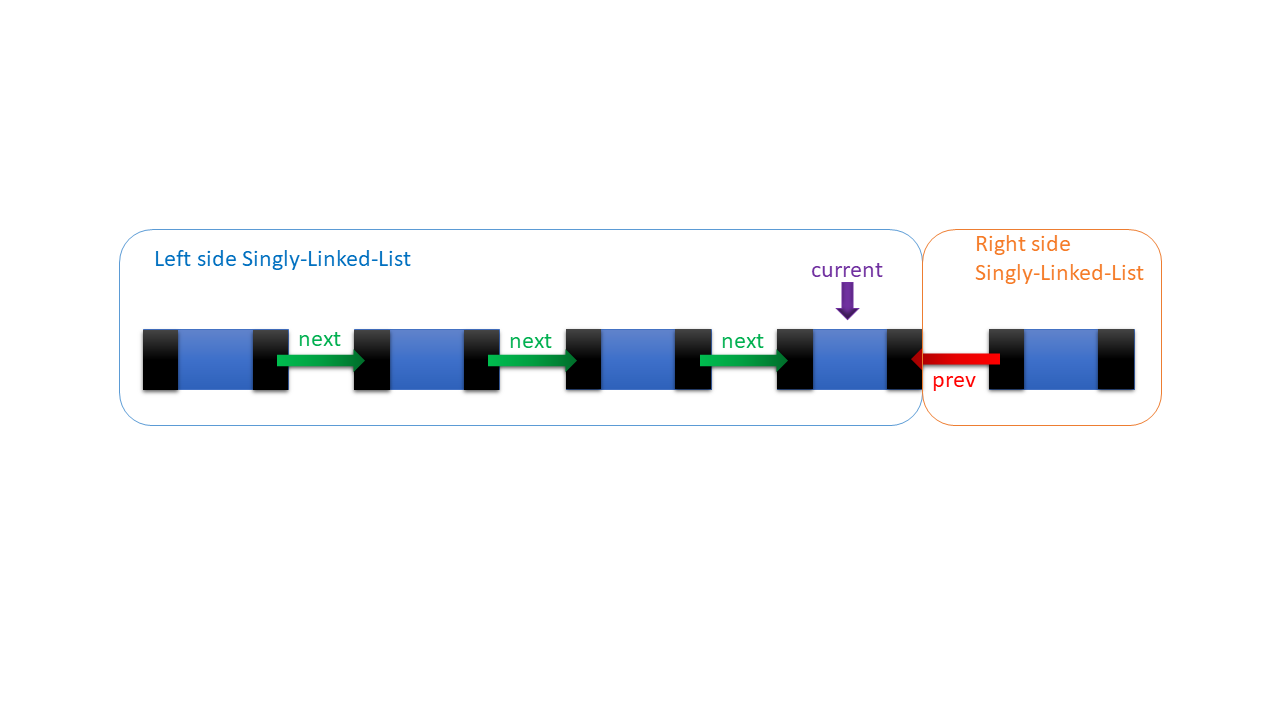
* In order to **navigate left**, we will pop the last element from the Left side Singly-Linked-List and push it to the Right side Singly-Linked-List.
* In order to **navigate right**, we will pop the last element from the Right side Singly-Linked-List and push it to the Left side Singly-Linked-List.

We will use move semantics of Rust to do that. And it will be completely safe!!!

From the current stage we can move to the left:



Or move to the right:



Voilà! We managed to maintain the functionality of the data structure and developed a safer model for Doubly-Linked-List. Now we can implement it and add awesome functionality to our structure, which will make that efficient, safe, and reliable.

# Implementing Doubly Linked List in Rust

Finally! We reached the implementation stage! In order to make our code structured, we have to use the module system of Rust. We will create a module for Singly-Linked-List and Doubly-Linked-List. Our File Structure will be like that:

src

├── main.rs

├── doubly\_linked\_list

│   └── list.rs

└── doubly\_linked\_list.rs

Singly-Linked-List structure module (list.rs) will be included in the Doubly-Linked-List structure module (doubly\_linked\_list.rs), Doubly-Linked-List module will be included in the main file (main.rs). Note, Box Pointers always be wrapped by Option, because there is no NULL pointer in Rust, but Option provides us None.

## Implementation of Singly-Linked-List

In order to create Singly-Linked-List, we will need two structures, one for Singly-Linked-List itself and another for the nodes. Let’s create a structure for Singly-Linked-List:

*pub struct* List<T> {  
 head: Option<Box<Node<T>>>,  
 size: *i32*,  
}

List structure will have two keys:

* head: pointer to the last node of the Singly-Linked-List
* size: 32-bit integer for the size of the Singly-Linked-List

Node Structure will look like that:

*pub struct* Node<T> {  
 elem: T,  
 next: Option<Box<Node<T>>>,  
}

* elem: for the value resided into the node
* next: for the pointer to the next node

As we defined the structures, we can implement the methods for List<T>. The first step is to create define the function which will return empty instance for the structure:

*impl*<T> List<T> {  
 *pub fn new*() -> *Self* {  
 List {  
 head: *None*,  
 size: 0,  
 }  
 }  
}

the function returns empty instance with *None* stored in head and 0 in size.

We can test our code writing ‘#[test]’ to the tester function and executing ‘cargo test’ command:

Test:

#[test]  
*fn* create\_new() {  
 *let* list: List<*i32*> = List::*new*();  
 assert\_eq!(list, List {  
 head: *None*,  
 size: 0,  
 } *as* List<*i32*>);  
}

Result:

C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\src\doubly\_linked\_list>cargo test  
error[E0369]: binary operation `==` cannot be applied to *type* `doubly\_linked\_list::list::List<i32>`  
--> src\doubly\_linked\_list\list.rs:142:5  
|  
142 | assert\_eq!(list, List {  
 | \_\_\_\_\_^  
 | |\_\_\_\_\_|  
 | |  
143 | | head: None,  
144 | | size: 0,  
145 | | } *as* List<i32>);  
| | ^  
| |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|  
| |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_doubly\_linked\_list::list::List<i32>  
| doubly\_linked\_list::list::List<i32>  
|  
= note: an implementation of `std::cmp::PartialEq` might be missing *for* `doubly\_linked\_list::list::List<i32>`  
= note: this error originates *in* a *macro* (*in* Nightly builds, run with -Z *macro*-backtrace *for* more info)

We got this error because our struct doesn’t support ‘==’ operator, to make the mentioned operator supported, we just need to use std::cmp, write #[derive(PartialEq)] above the declared structures. There is no need to write operator functions because Rust will automatically derive functionality for the List<T> and Node<T>.

Let’s add it to the Structures and run it again!

#[derive(PartialEq)]  
*pub struct* List<T> {  
 head: Option<Box<Node<T>>>,  
 size: *i32*,  
}  
  
#[derive(PartialEq)]  
*pub struct* Node<T> {  
 elem: T,  
 next: Option<Box<Node<T>>>,  
}

Test Result:

C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\src\doubly\_linked\_list>cargo test  
Compiling doubly-linked-list v0.1.0 (C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list)  
Finished test [unoptimized + debuginfo] target(s) *in* 1.44s  
Running C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\target\debug\deps\doubly\_linked\_list-f4dd80bb82e872e1.exe  
  
running 1 test  
test doubly\_linked\_list::list::create\_new ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

### List::push()

In order to insert elements in the list, push() function has to be created.

List::*push*() must implement following actions:

* Take a new element to push it in the List
* Create Box Pointer to the new Node
* Next key of New node has to be the Box Pointer of the last element of the List
* head of the List has to be Box Pointer to the new Node
* size of the List has to be increased by 1

*impl*<T> List<T> {  
 *pub fn* push(&*mut self*, elem: T) {  
 *self*.head = *Some*(Box::*new*(Node { elem: elem, next: *self*.head.take() }));  
 *self*.size += 1;  
 }  
}

Note that, in Node, next is *self*.head.take(). We need to use .take() to move value from *self*.head and store None in *self*.head.

* The value of *self*.head will be replaced by None.
* The value taken from *self*.head will be moved to the next key of the new Node

### List::pop()

List::*pop*() function will be used for popping and taking the moved out element from the List.

List::*pop*() must implement following actions:

* Return the element of the Node stored in the head
* head of the List has to be Box Pointer to the next value of Node
* size of the List has to be decreased by 1
* if there is nothing to pop, *None* will be returned

*impl*<T> List<T> {  
 *pub fn* pop(&*mut self*) -> Option<T> {  
 *self*.size = cmp::max(*self*.size - 1, 0);  
 *match self*.head.take() {  
 *Some*(node) => {  
 *self*.head = node.next;  
 *Some*(node.elem)  
 }  
 *None* => *None*,  
 }  
 }  
}

Let’s Test our functions described above:

Test:

#[test]  
*fn* push\_and\_pop(){  
 *let mut* list: List<*i32*> = List::*new*();  
 list.push(1);  
 list.push(2);  
 list.push(3);  
 assert\_eq!(list.pop(),*Some*(3));  
 assert\_eq!(list.pop(),*Some*(2));  
 assert\_eq!(list.pop(),*Some*(1));  
 assert\_eq!(list.pop(),*None*);  
}

Result:

C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\src\doubly\_linked\_list>cargo test  
Compiling doubly-linked-list v0.1.0 (C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list)  
Finished test [unoptimized + debuginfo] target(s) *in* 4.58s  
Running C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\target\debug\deps\doubly\_linked\_list-f4dd80bb82e872e1.exe  
  
running 2 tests  
test doubly\_linked\_list::list::create\_new ... ok  
test doubly\_linked\_list::list::push\_and\_pop ... ok  
  
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Everything works fine!

In order to get the head and size of the List, we have to define two getter functions for them.

### List::get\_size()

List::*get\_size*() must implement following actions:

* return the size of List

*impl*<T> List<T> {  
 *pub fn* get\_size(&*self*) -> *i32* {  
 *self*.size  
 }  
}

### List::get\_top()

List::*get\_top*() must implement following actions:

* return the reference of node element stored in the head of List

*impl*<T> List<T> {  
 *pub fn* get\_top(&*self*) -> Option<&T> {  
 *match self*.head.as\_ref() {  
 *Some*(node) => {  
 *Some*(&node.elem)  
 }  
 *None* => *None* }  
 }  
}

Lets check if getters work:

Test:

#[test]  
*fn* getters(){  
 *let mut* list : List<*i32*> = List::*new*();  
 list.push(33);  
 list.push(22);  
 list.push(11);  
  
 assert\_eq!(list.get\_size(),3);  
 assert\_eq!(list.get\_top(),*Some*(&11));  
}

Result:

C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\src\doubly\_linked\_list>cargo test  
Compiling doubly-linked-list v0.1.0 (C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list)  
Finished test [unoptimized + debuginfo] target(s) *in* 1.44s  
Running C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\target\debug\deps\doubly\_linked\_list-f4dd80bb82e872e1.exe  
  
running 3 tests  
test doubly\_linked\_list::list::getters ... ok  
test doubly\_linked\_list::list::push\_and\_pop ... ok  
test doubly\_linked\_list::list::create\_new ... ok  
  
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

### Iterator

Nice! In order to make our Singly-Liked-List better than a stack, we have to make it iterable. We should have a *trait*. Traits are just like a sketch of the functions and properties that can be applied to any structure.

*trait Iterator* {  
 *type* Item;  
 *fn* next(&*mut self*) -> Option<*Self*::Item>;  
}

In Item, we will save generic type (with lifetime parameter) of the List<T>.

Now, Let’s create a structure called Iter, that will traverse over List<T>.

*pub struct* Iter<'a, T> {  
 next: Option<&'a Node<T>>,  
}

Now, iter should be implemented for List<T>. We have to borrow the reference of the node and store it in Iter. In order to convert &Option<T> to Option<&T>, as\_ref() have to be used. As we borrow the reference of Box pointer, we have to dereference borrowed reference of value, then dereference Box pointer to get the Node<T> from &Box<Node<T>>. After we get the Node<T> we have to return the reference of it, so two ‘\*’ for getting the Node<T> and one ‘&’ for returning a reference of it.

*impl*<T> List<T> {

*pub fn* iter(&*self*) -> Iter<T> {  
 Iter {  
 next: *self*.head.as\_ref().map(|node:&Box<Node<T>>| { &\*\*node }),  
 }  
 }  
}

Created Iterator trait has to be applied to the Linked list. Item of the iterator will be a structure with an independent lifetime, therefore references, type-generics, and lifetime parameters will be used.

next() function of Iterator has to return Option, having reference of value stored in the node. To make that:

* Iter should store reference of the next node
* Reference of current node element (wrapped with Option) have to be returned

*impl*<'a, T> *Iterator for* Iter<'a, T> {  
 *type* Item = &'a T;  
  
 *fn* next(&*mut self*) -> Option<*Self*::Item> {  
 *self*.next.map(|node| {  
 *self*.next = node.next.as\_ref().map(|next\_node| { &\*\*next\_node });  
 &node.elem  
 })  
 }  
}

It’s time to test:

#[test]  
*fn* list\_iter(){  
 *let mut* list : List<*i32*> = List::*new*();  
 list.push(33);  
 list.push(22);  
 list.push(11);  
 *let mut* it : Iter<*i32*> = list.iter();  
 assert\_eq!(it.next(),*Some*(&11));  
 assert\_eq!(it.next(),*Some*(&22));  
 assert\_eq!(it.next(),*Some*(&33));  
 *//Iterator has no influence on the list* assert\_eq!(list.get\_top(),*Some*(&11));  
}

Result:

C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\src\doubly\_linked\_list>cargo test  
Compiling doubly-linked-list v0.1.0 (C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list)  
Finished test [unoptimized + debuginfo] target(s) *in* 1.12s  
Running C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\target\debug\deps\doubly\_linked\_list-f4dd80bb82e872e1.exe  
  
running 4 tests  
test doubly\_linked\_list::list::create\_new ... ok  
test doubly\_linked\_list::list::getters ... ok  
test doubly\_linked\_list::list::list\_iter ... ok  
test doubly\_linked\_list::list::push\_and\_pop ... ok  
  
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

To implement a mutable Iterator, we just need to use &mut instead of mut and as\_mut instead of as\_ref. So Struct, function, and iterator implementation will look like that:

Struct:

*pub struct* IterMut<'a, T> {  
 next: Option<&'a *mut* Node<T>>,  
}

Function:

*impl*<T> List<T> {  
 *pub fn* iter\_mut(&*mut self*) -> IterMut<T> {  
 IterMut {  
 next: *self*.head.as\_mut().map(|node| { &*mut* \*\*node }),  
 }  
 }  
}

Iterator on IterMut:

*impl*<'a, T> *Iterator for* IterMut<'a, T> {  
 *type* Item = &'a *mut* T;  
  
 *fn* next(&*mut self*) -> Option<*Self*::Item> {  
 *self*.next.take().map(|node| {  
 *self*.next = node.next.as\_mut().map(|next\_node| { &*mut* \*\*next\_node });  
 &*mut* node.elem  
 })  
 }  
}

test:

#[test]  
*fn* list\_iter\_mut(){  
 *let mut* list : List<*i32*> = List::*new*();  
 list.push(33);  
 list.push(22);  
 list.push(11);  
 *let mut* it : IterMut<*i32*> = list.iter\_mut();  
 assert\_eq!(it.next(),*Some*(&*mut* 11));  
 assert\_eq!(it.next(),*Some*(&*mut* 22));  
 assert\_eq!(it.next(),*Some*(&*mut* 33));  
 *//Iterator has no influence to the list* assert\_eq!(list.get\_top(),*Some*(&11));  
}

Result:

C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\src\doubly\_linked\_list>cargo test  
Compiling doubly-linked-list v0.1.0 (C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list)  
Finished test [unoptimized + debuginfo] target(s) *in* 1.43s  
Running C:\Users\Sandro\Desktop\RUST\Rust\doubly-linked-list\target\debug\deps\doubly\_linked\_list-f4dd80bb82e872e1.exe  
  
running 5 tests  
test doubly\_linked\_list::list::getters ... ok  
test doubly\_linked\_list::list::list\_iter\_mut ... ok  
test doubly\_linked\_list::list::list\_iter ... ok  
test doubly\_linked\_list::list::create\_new ... ok  
test doubly\_linked\_list::list::push\_and\_pop ... ok  
  
test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

### Drop

As destructors in Rust, *Drop* *trait* is used. We have to implement it to make our memory free, after the usage of List<T>.

*Drop* *trait* will have only one function, and we call it drop():

*pub trait Drop* {  
 *fn* drop(&*mut self*);  
}

In the drop function, we have to take value from the current node and get on the next node until we don’t get None in the variable.

*impl*<T> *Drop for* List<T> {  
 *fn* drop(&*mut self*) {  
 *let mut* current: Option<Box<Node<T>>> = *self*.head.take();  
 *while let Some*(*mut* boxed\_node) = current {  
 current = boxed\_node.next.take();  
 }  
 }  
}

*while let* will work until we don’t get *None* into the current. As a result, all the value will be moved out of the Singly-Linked-List, and memory will be freed out.

## Implementation of Doubly-Linked-List

We’ve reached the final and the most decisive part of our project. We have to use all of our knowledge and implemented code, in order to create a sophisticated data structure.

Following modules will be used:

*use* std::{cmp, mem};  
*use* std::fmt::*Debug*;  
  
*mod* list;  
  
*use* list::List;  
*use crate*::doubly\_linked\_list::list::*Drop as* list\_drop;

Let’s define the structure:

*pub struct* DoublyLinkedList<T> {  
 left: List<T>,  
 right: List<T>,  
 size: *i32*,  
}

As we showed in explanation Doubly-Linked-List will have left side Singly-Linked-List and right side Singly-Linked-List, the size will be an integer which will store the quantity of the elements stored in Doubly-Linked-List.

Let’s take the initial step and start with a method which returns the instance of the structure:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn new*() -> *Self* {  
 *let mut* left\_list = List::*new*();  
 *let mut* right\_list = List::*new*();  
 DoublyLinkedList {  
 left: left\_list,  
 right: right\_list,  
 size: 0,  
 }  
 }  
}

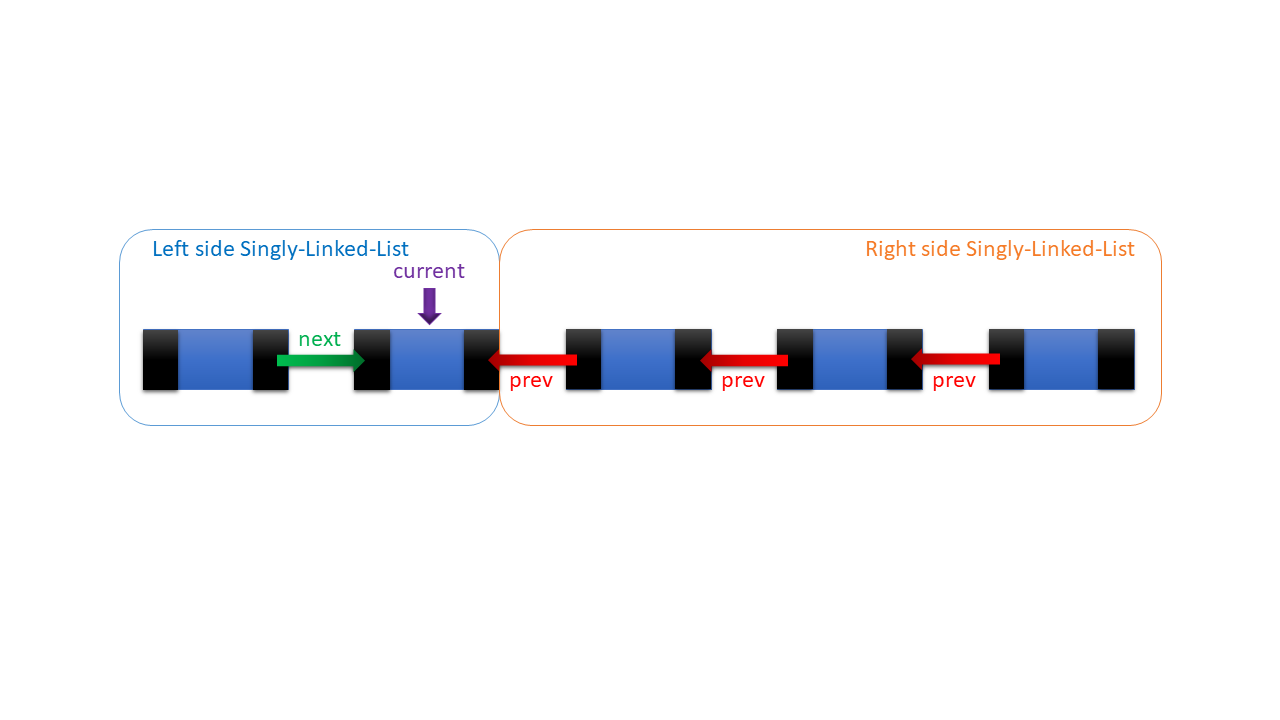
Left side and right side lists are created and moved in the Doubly-Linked-List. After that new struct is returned.

Let’s create getter 2 functions for size, one will return the size wrapped in Option, and the second will return the size as an integer:

*pub fn* get\_size(&*self*) -> Option<&*i32*> {  
 *Some*(&*self*.size)  
}  
  
*pub fn* size(&*self*) -> *i32* {  
 \**self*.get\_size().unwrap()  
}

These functions will be used by other methods of the data structure.

Let look at the visualization of the Doubly-Linked-List again.



We are calling the element current if it is the last element of left side Singly-Linked-List so create getter functions that implement this concept.

*pub fn* get\_current(&*mut self*) -> Option<&T> {  
 *self*.left.get\_top()  
}  
  
*pub fn* get\_current\_position(&*mut self*) -> *i32* {  
 *self*.left.get\_size() - 1  
}

* The current element is the last element of left side Singly-Linked-List
* Current Position in Doubly-Linked-List is the number of elements stored in left side Singly-Linked-List minus one because we start indexing from 0

For the debugging reasons we have to create checker methods, that will prevent us from the errors.

Let’s define a function which checks if the Doubly-Linked-List is empty:

*impl*<T> DoublyLinkedList<T> {  
 *fn* empty(&*self*) -> *bool* {  
 *let* is\_empty: *bool* = *self*.size() == 0;  
 is\_empty  
 }  
}

As we can check emptiness easily, we can create a function which will panic if the Doubly-Linked-List is empty:

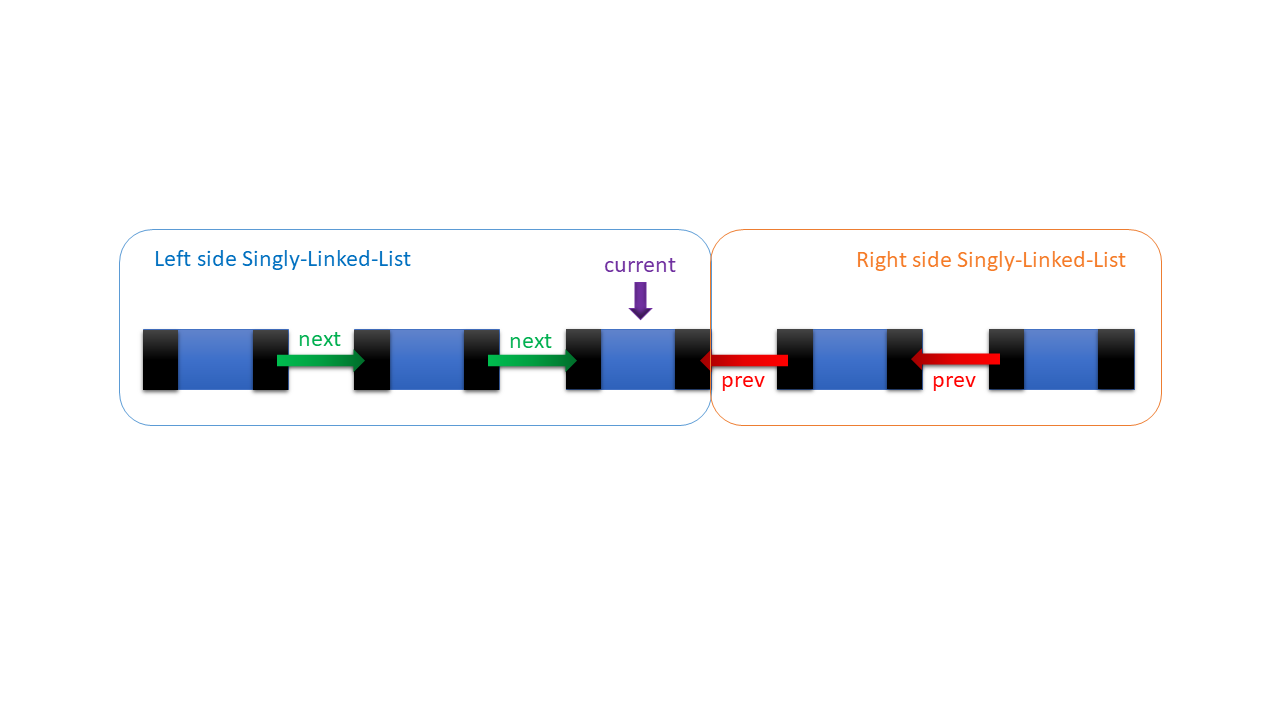
*impl*<T> DoublyLinkedList<T> {  
 *fn* check\_empty(&*mut self*) {  
 *if* (*self*.empty()) {  
 panic!("Doubly-Linked-List is Empty!!!");  
 }  
 }  
}

We will also need a function which checks if an index is in the range of positions of Doubly-Linked-List:

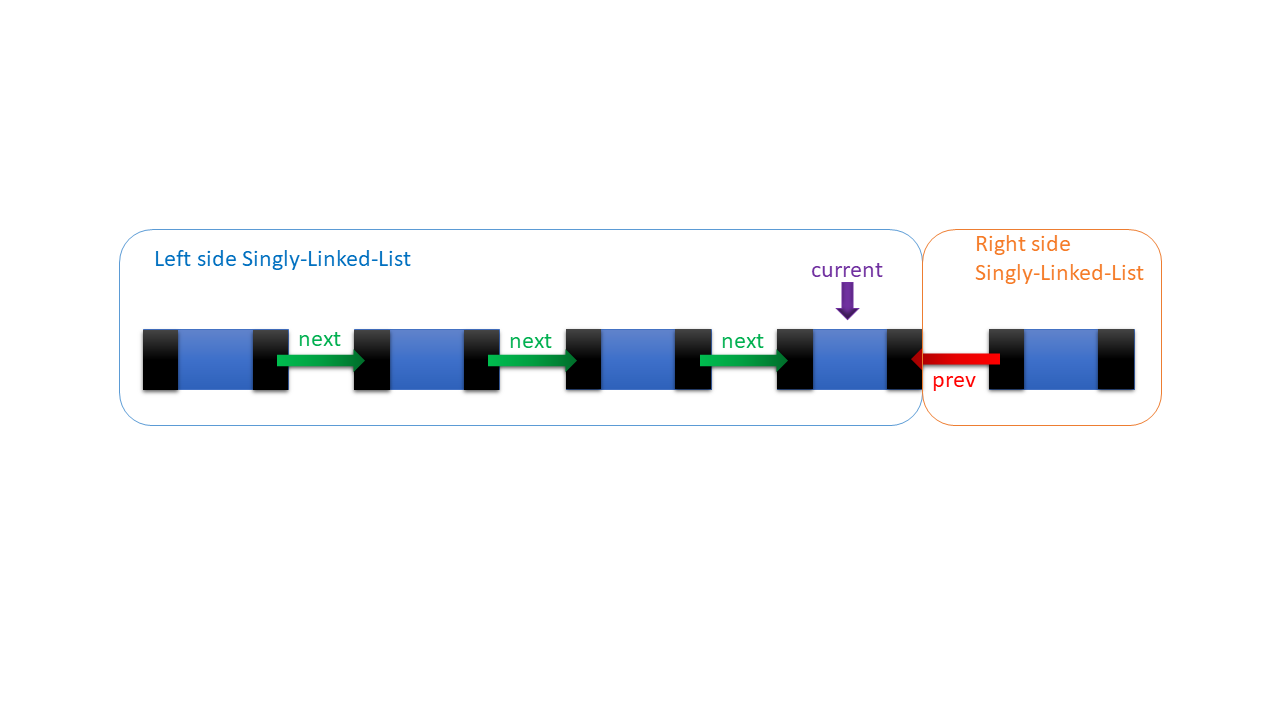
*impl*<T> DoublyLinkedList<T> {  
 *fn* check\_size(&*mut self*, index: *i32*) {  
 *self*.check\_empty();  
 *if* (*self*.size() <= index || index < 0) {  
 panic!("Index Out of Bounds!!!");  
 }  
 }  
}

Since we have getter and checker functions, we can create functions that will change make Doubly-Linked-List functional.

In order to change the current position to the right, element has to move from the **right side Singly-Linked-List** to the **left side Singly-Linked-List**:



After moving right:

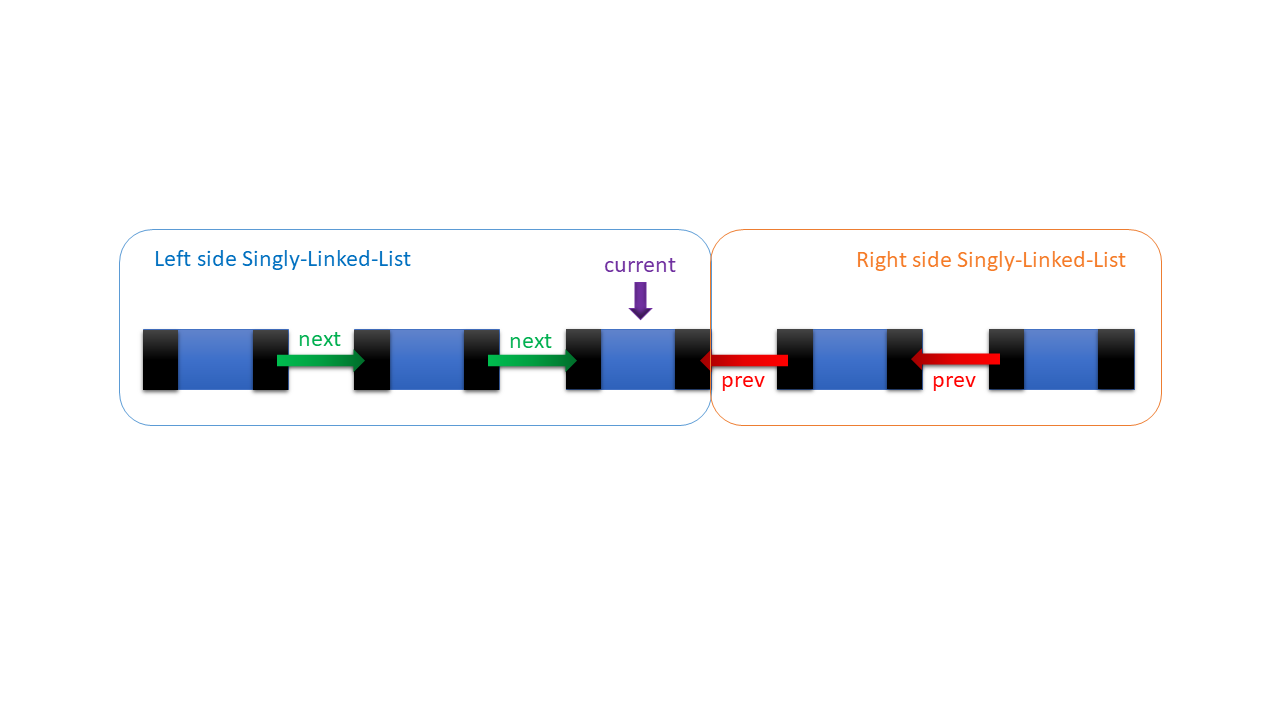


All we need to do is popping from the right and pushing to left:

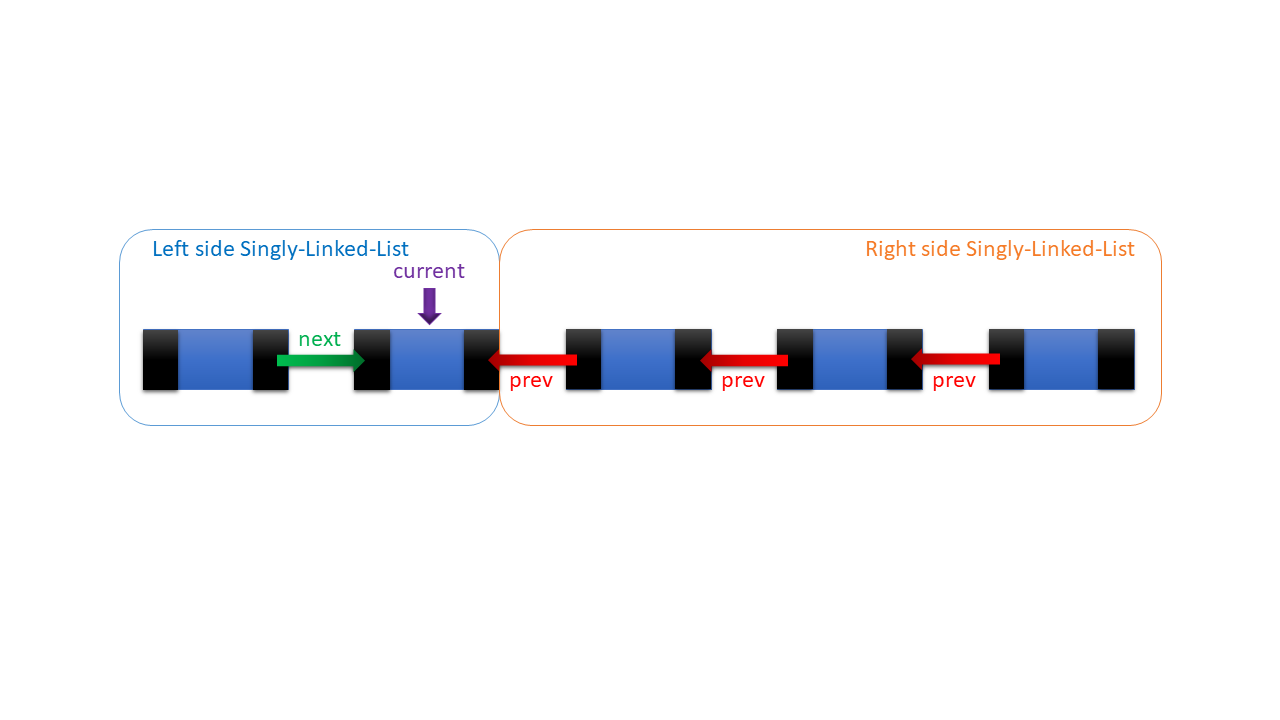
*impl*<T> DoublyLinkedList<T> {  
 *pub fn* next(&*mut self*) {  
 *if* (*self*.right.get\_size() != 0) {  
 *self*.right.pop().map(|node| { *self*.left.push(node); });  
 }  
 }  
}

note that: get\_size() is the method of right side Singly-Linked-List, it doesn’t belong to the Doubly-Linked-List.

In order to change the current position to the left, element has to move from the **left side Singly-Linked-List** to the **right side Singly-Linked-List**:



After moving left:



All we need to do is popping from left and pushing to right:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* previous(&*mut self*) {  
 *if* (*self*.left.get\_size() != 0) {  
 *self*.left.pop().map(|node| { *self*.right.push(node); });  
 }  
 }  
}

Since we can navigate through Doubly-Linked-List, we have to create functions that add new data to the structure.

At first, let’s define the method, which inserts a new element at the end(rightmost point) of the Doubly-Linked-List:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* push\_back(&*mut self*, elem: T) {  
 *while* (*self*.right.get\_size() != 0) {  
 *self*.next();  
 }  
 *self*.left.push(elem);  
 *self*.size += 1;  
 }  
}

At the initial stage, we are move to end using *self*.next() function. Since the right side Singly-Linked-List became empty, we pushed new element to the left side Singly-Linked-List and increased the size of Doubly-Linked-List.

It would be convenient if we had a function, which set the current element at any position. So, we can create a function which changes (shifts) current position to the desired position:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* shift(&*mut self*, index: *i32*) {  
 *while* (*self*.get\_current\_position() != index) {  
 *if* (index > *self*.get\_current\_position()) {  
 *self*.next();  
 } *else* {  
 *self*.previous();  
 }  
 }  
 }  
}

If the desired position is more than the current position, it shifts to right using *self*.next() function.

If the desired position is less than the current position, it shifts to left using *self*.previous() function.

Given that, we can easily implement the function, which inserts a new element at any existing position:

*pub fn* push(&*mut self*, elem: T, index: *i32*) {  
 *if*(index == *self*.size()){  
 *self*.shift(index - 1);  
 *self*.right.push(elem);  
 }  
 *else* {  
 *self*.check\_size(index);  
 *self*.shift(index - 1);  
 *self*.left.push(elem);  
 }  
 *self*.size += 1;  
}

function checks if the index is in an existing position, then the current position is shifted, the element is pushed at a desirable position, and the size of Doubly-Linked-List is increased. If the passed index is equal to the size of Doubly-Linked-List it will work as push\_back(), right side Singly-Linked-List will be emptied and the new element will be pushed to the right.

Since we have insertion functions, we also need deletion functions. Let’s define the function which erases the rightmost element from the data structure:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* pop\_back(&*mut self*) {  
 *self*.shift(*self*.size() - 1);  
 *self*.left.pop();  
 *self*.size = cmp::max(0, *self*.size - 1);  
 }  
}

the function will shift the current element to the rightmost position, the last element will be popped and size will be decreased by 1, though the value of *self*.size never becomes less than 0.

To make deletion functionality better, let’s add a function, which will erase an element at a certain position:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* pop(&*mut self*, index: *i32*) {  
 *self*.check\_size(index);  
 *self*.shift(index);  
 *self*.left.pop();  
 *self*.size = cmp::max(0, *self*.size - 1);  
 }  
}

To get an element from the Doubly-Linked-List, get() function will be declared:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* get(&*mut self*, index: *i32*) -> &T {  
 *self*.check\_size(index);  
 *self*.shift(index);  
 *self*.left.get\_top().unwrap()  
 }  
}

We will check if the required position is in the range of available positions, the current position will be shifted to desiring position, and reference of the current element will be returned from the left side Singly-Linked-List.

As we can read, create and delete the data, update functions also have to be implemented let’s implement update function for current element:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* edit\_current(&*mut self*, value: T) {  
 *self*.left.pop();  
 *self*.left.push(value);  
 }  
}

the current element will be popped from the left side Singly-Linked-List and the new element will be placed at the current position.

For convenience, user have to be able to update an element at the desired position, therefore we have to implement the following function:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* edit(&*mut self*, index: *i32*, value: T) {  
 *self*.check\_size(index);  
 *self*.shift(index);  
 *self*.left.pop();  
 *self*.left.push(value);  
 }  
}

The function will check if the passed index argument fits the range, then the current value will be shifted to the favored position, the current element will be popped from left side Singly-Linked-List and the new value will be pushed at the current position.

Since we are able to create, read, update, and delete the data of Doubly-Linked-List, we can go beyond basic functionality and implement sort function. We will implement a merge sort algorithm, which will work in .

To implement it we have to split data into the segments, length of the exponent of 2. After splitting, segments will be merged into the sorted order.

Sorting function will be bigger compared to others therefore we will explain it line by line, unit by unit:

Declaration of function:

*pub fn* sort(&*mut self*)  
 *where* T: std::cmp::*PartialEq* + std::cmp::*PartialOrd*{  
 *//some code here*}

*where* T: std::cmp::*PartialEq* + std::cmp::*PartialOrd*: it means that Type (variable of type T) have to implement *PartialEq* and *PartialOrd* traits, because we have to be able to compare element in order to sort.

Taking the corner case into perspective:

*pub fn* sort(&*mut self*)  
 *where* T: std::cmp::*PartialEq* + std::cmp::*PartialOrd*{  
 *if* (*self*.empty()) {  
 *return*;  
 }  
}

If Doubly-Linked-List empty and there is nothing to sort then we just return doing nothing.

Declaration of variables:

*pub fn* sort(&*mut self*)  
 *where* T: std::cmp::*PartialEq* + std::cmp::*PartialOrd*{  
 *if* (*self*.empty()) {  
 *return*;  
 }  
 *self*.shift(0);  
 *let mut* exp = 2;  
 *let mut* buffer: List<T> = List::*new*();  
 *let* size = *self*.size();  
}

*self*.shift(0) : Our current element has to be shifted to 0 to start sorting from the first index.

*let mut* exp = 2: exponent of 2 will be increased as the size of a segment to sort.

*let mut* buffer: List<T> = List::*new*(): it will be used as temporary storage, which will help us to sort segments.

*let* size = *self*.size(): during the execution of the sort function size of the list remains constant so size doesn’t need to be mutable.

*pub fn* sort(&*mut self*)  
 *where* T: std::cmp::*PartialEq* + std::cmp::*PartialOrd*{  
 *if* (*self*.empty()) {  
 *return*;  
 }  
 *self*.shift(0);  
 *let mut* exp = 2;  
 *let mut* buffer: List<T> = List::*new*();  
 *let* size = *self*.size();  
 *while* ((exp >> 1) < size) {  
 *//some code that will sort exp size segments* exp = exp << 1;  
 }  
}

(exp >> 1): literally means (exp / 2), that’s a bitwise shift to right, as we shift binary number to the right, it will be reduced by two times.

exp = exp << 1: literally means exp = exp \* 2, that’s a bitwise shift to left, as we shift binary number to the left, it will become two times bigger.

Bitwise operations are faster and show better in code how the exponent variable is growing on each iteration.

If the middle index of a segment (exp >> 1) becomes greater than the last index of Doubly-Linked-List, *while*-loop will stop.

*while* ((exp >> 1) < size) {  
 *let* init = (exp >> 1) - 1;  
 *self*.shift(init);  
 *for* idx *in* init..size {  
 *//some code here* }  
 exp = exp << 1;  
}

*let* init = (exp >> 1) - 1: as initial index we take the middle index of segment reduced by one since we start indexing from 0. On each iteration, we are inferring that all elements left to the init are already sorted.

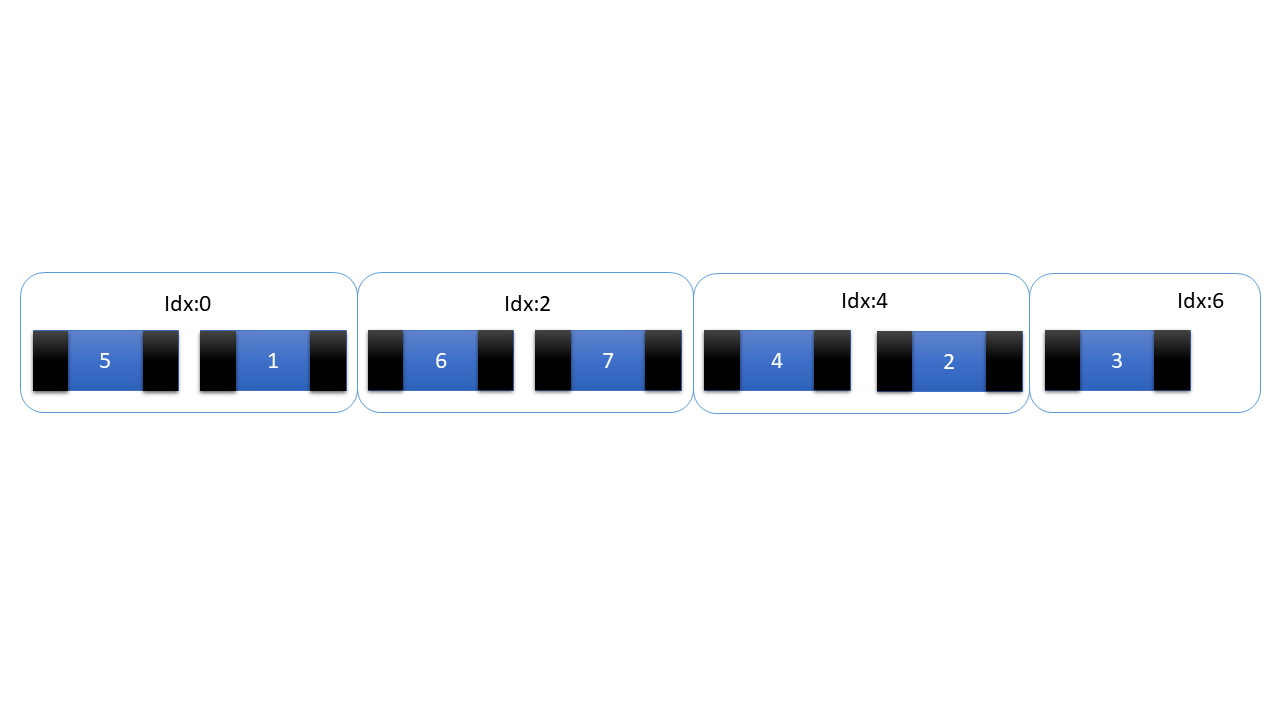
*self*.shift(init) : we are shifting the current element to the initial index.

*for* idx *in* init..size: iterating over Doubly-Linked-List.

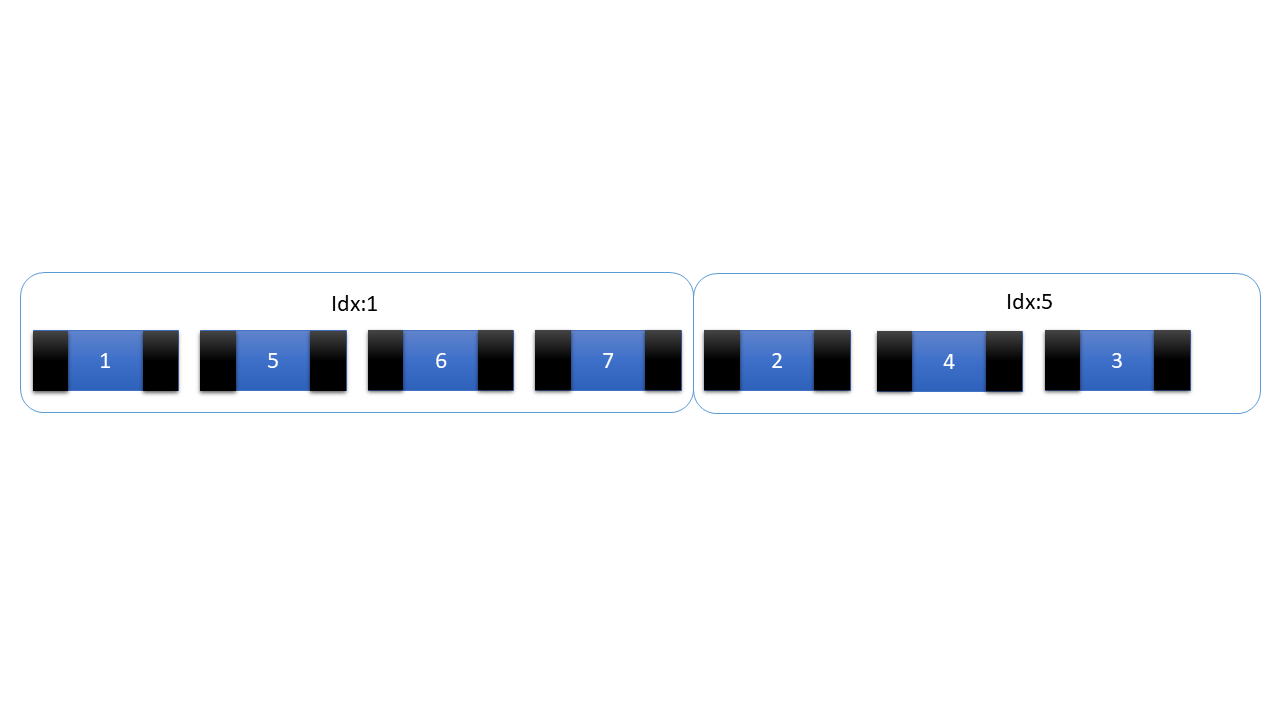
*for* idx *in* init..size {  
 *self*.shift(idx);  
 *if* ((idx - init) % exp == 0) {  
 *// some code here* }  
}

*self*.shift(idx) : On each iteration current element will be shifted to the current index idx.

(idx - init) % exp == 0: This formula gives true when we are standing on proper index of a segment.

init:0, exp:2, true for indexes: 0, 2, 4, 6, 8…

init:1, exp:4, true for indexes: 1, 5, 9, 13…



Note that: after first iteration 5 -> 3 become 3 -> 5.

*if* ((idx - init) % exp == 0) {  
 {  
 *let mut* left\_element = *self*.left.pop();  
 *let mut* right\_element = *self*.right.pop();  
 *let mut* half\_segment\_length = (exp >> 1);  
 *while* (right\_element != *None* &&

left\_element >= right\_element &&

half\_segment\_length > 0)

{  
 *//some code here* }  
 }  
}

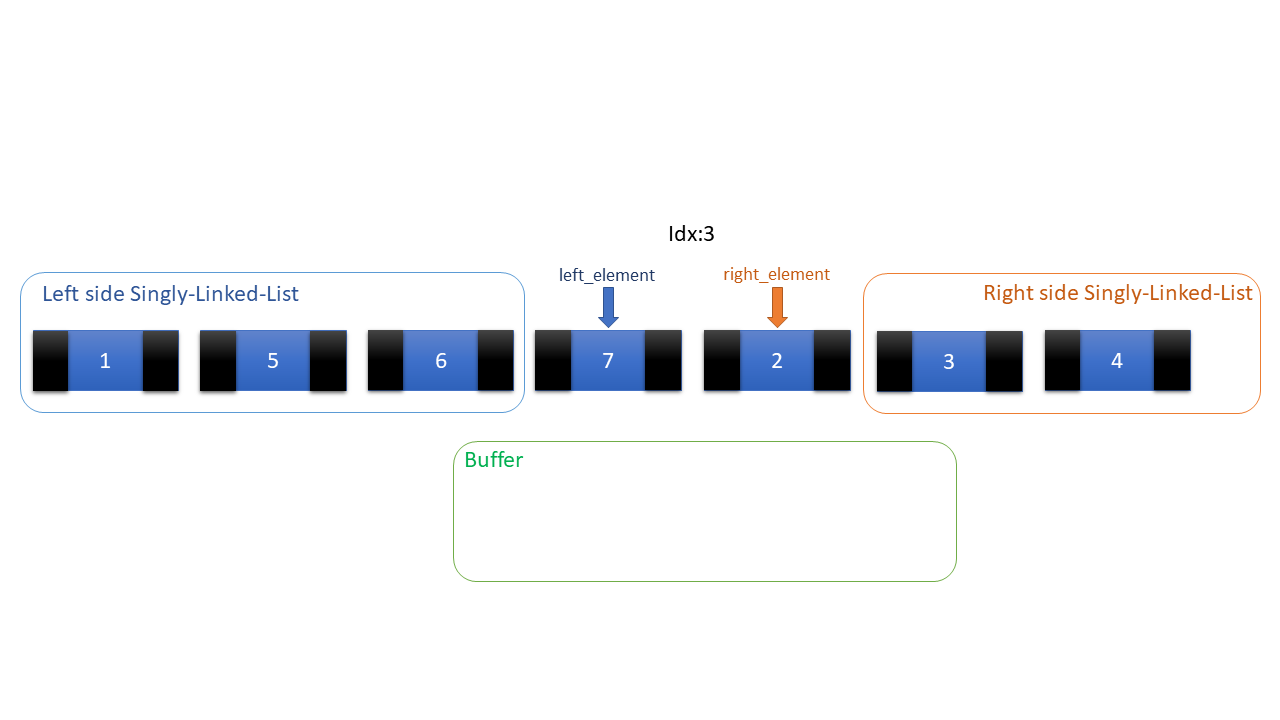
*let mut* left\_element = *self*.left.pop(): taking the last element from left side Singly-Linked-List to be compared to the values stored in the right side Singly-Linked-List.

*let mut* right\_element = *self*.right.pop(): taking the last element from right side Singly-Linked-List to be compared to the values stored in the left side Singly-Linked-List.

*let mut* half\_segment\_length = (exp >> 1): Since we are sorting only the segment of fixed size, we need this variable to make sure that we will not take more than required elements from left side Singly-Linked-List.

*while* (right\_element != *None* && left\_element >= right\_element && half\_segment\_length > 0): We will take elements from left side Singly-Linked-List and store them in buffer until we make sure that there is no element more than right\_element. If right\_element is *None* it means, right side Singly-Linked-List is empty and while loop will not be executed, because there is nothing to compare.

Init:3, exp:8, half\_segment\_length: 4



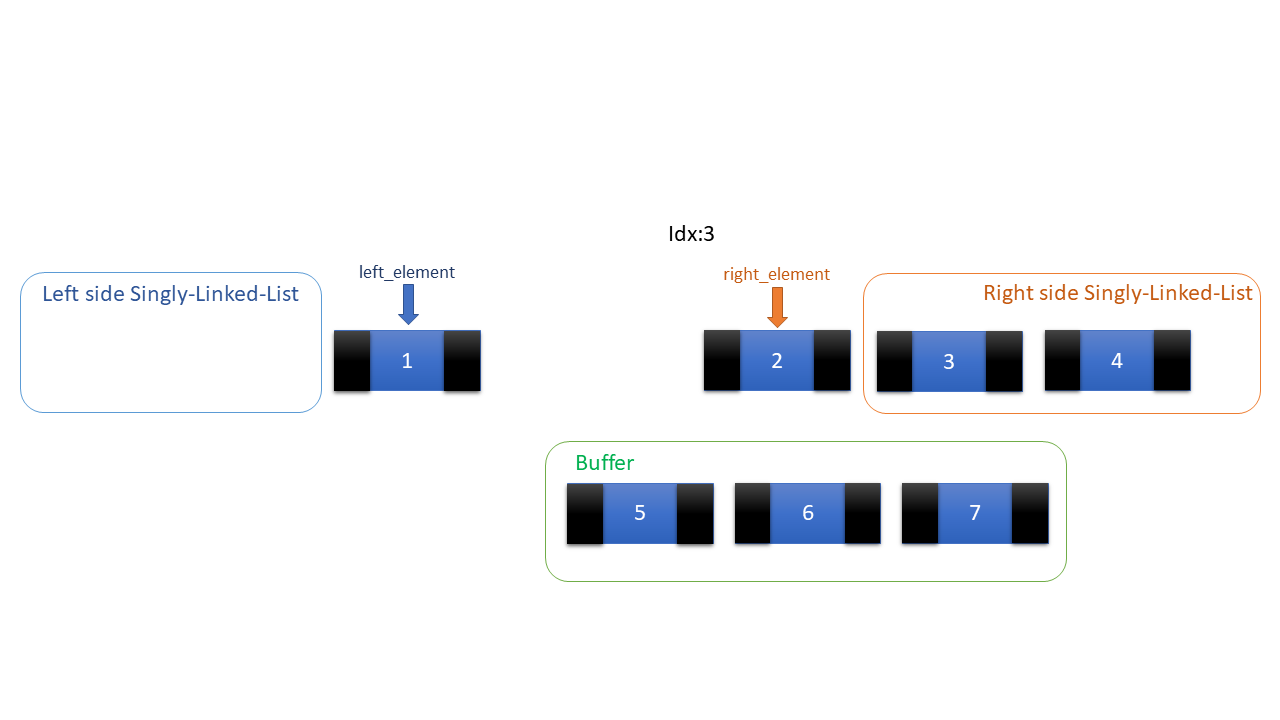
*while* (right\_element != *None* &&

left\_element >= right\_element &&

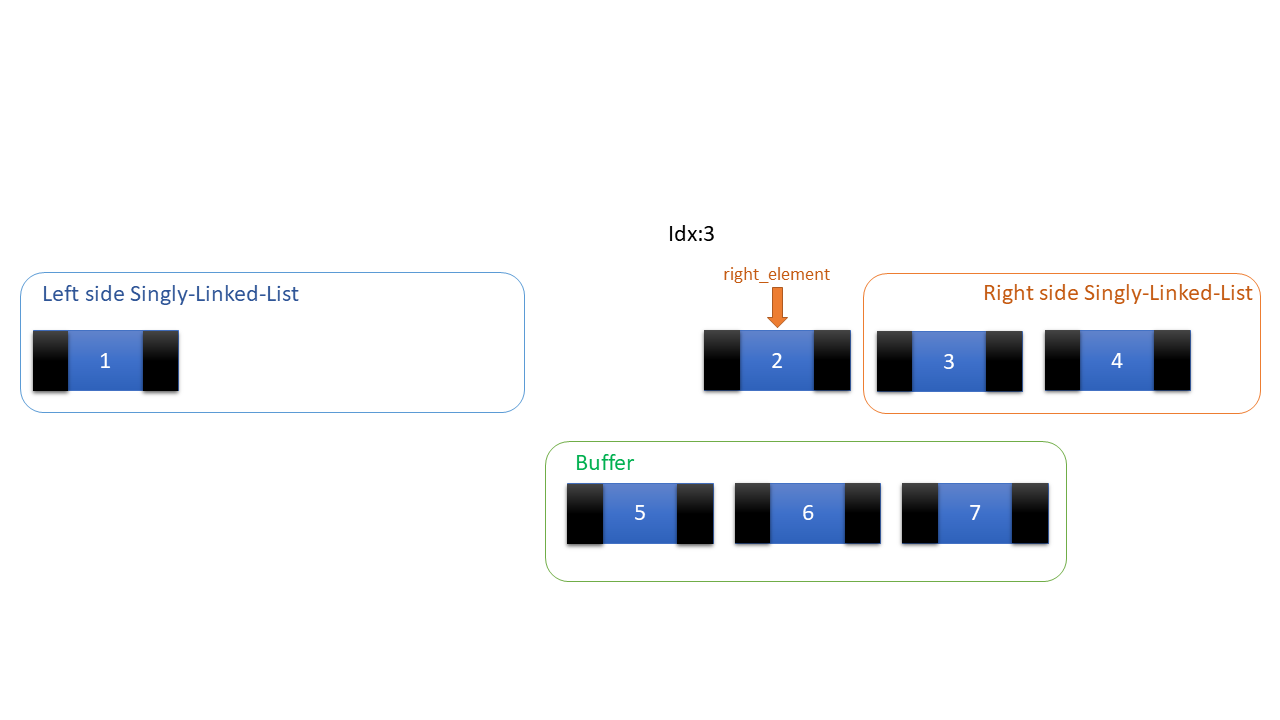
half\_segment\_length > 0)

{  
 *match* left\_element.take() {  
 *Some*(value) => {  
 buffer.push(value);  
 }  
 *None* => {  
 *break*;  
 }  
 }  
 left\_element = *self*.left.pop();  
 half\_segment\_length -= 1;  
}

Values more than right\_element will be moved into the buffer.

After the filling of the buffer, the last element taken by left\_element has to be returned to the left side Singly-Linked-List: 

*if* (left\_element != *None*) {  
 left\_element.take().map(|value| { *self*.left.push(value); });  
}



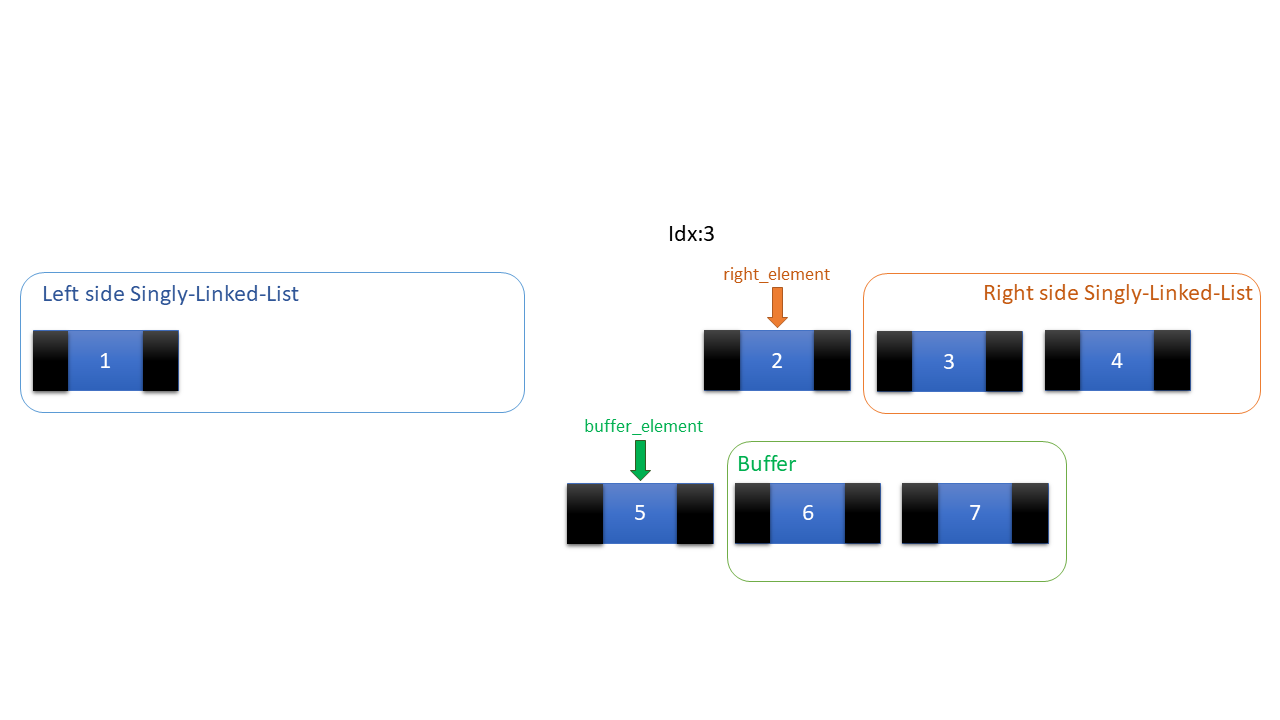
Now it’s time to merge buffer and right side Singly-Linked-List:

*let mut* buffer\_element = buffer.pop();  
half\_segment\_length = (exp >> 1);  
*while* (buffer\_element != *None*) {  
 *if* (half\_segment\_length > 0 &&

right\_element != *None* &&

right\_element < buffer\_element)

{  
 *match* right\_element.take() {  
 *Some*(value) => {  
 *self*.left.push(value);  
 }  
 *None* => {  
 *break*;  
 }  
 }  
 right\_element = *self*.right.pop();  
 half\_segment\_length -= 1;  
 } *else* {  
 *match* buffer\_element.take() {  
 *Some*(value) => {  
 *self*.left.push(value);  
 }  
 *None* => {  
 *break*;  
 }  
 }  
 buffer\_element = buffer.pop();  
 }  
}



*while* (buffer\_element != *None*): Until buffer is not empty, we will fill left side Singly-Linked-List.

*if* (half\_segment\_length > 0 &&

right\_element != *None* &&

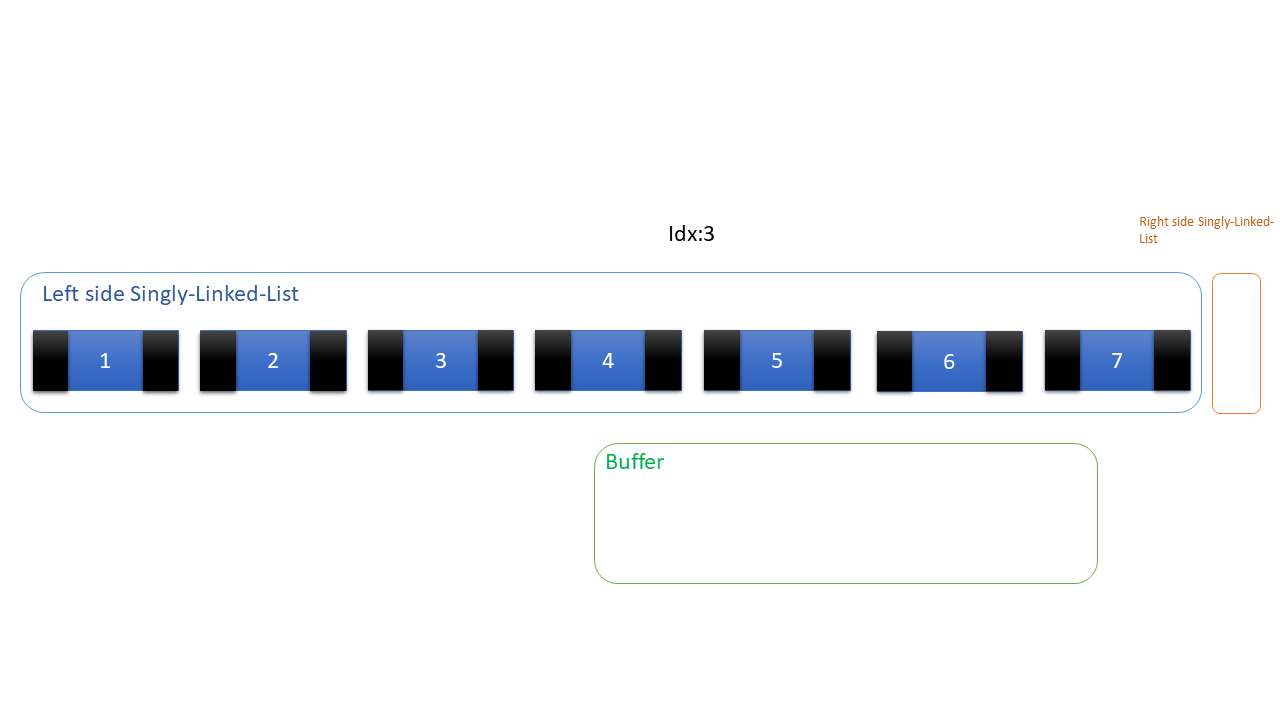
right\_element < buffer\_element) :

We are allowed to pop elements from right side Singly-Linked-List and push them to left side Singly-Linked-List, if half\_segment\_length limit is not exceeded, right hand side Singly-Linked-List is not empty and right\_element is less than buffer\_element.

If statements, mentioned above are not satisfied, then elements from buffer will be moved to left side Singly-Linked-List.

*if* (right\_element != *None*) {  
 right\_element.take().map(|value| { *self*.right.push(value); });  
}

Finally, we will check if something is taken by right\_element, we will return the value to the right side Singly-Linked-List.



Described processes will be executed, for all segments and in the end, we will get sorted Doubly-Linked-List.

Full code of sort function looks like that:

*impl*<T> DoublyLinkedList<T> {  
 *pub fn* sort(&*mut self*)  
 *where* T: std::cmp::*PartialEq* + std::cmp::*PartialOrd* {  
 *if* (*self*.empty()) {  
 *return*;  
 }  
 *self*.shift(0);  
 *let mut* exp = 2;  
 *let mut* buffer: List<T> = List::*new*();  
 *let* size = *self*.size();  
 *while* ((exp >> 1) < size) {  
 *let* init = (exp >> 1) - 1;  
 *self*.shift(init);  
 *for* idx *in* init..size {  
 *self*.shift(idx);  
 *if* ((idx - init) % exp == 0) {  
 {  
 *let mut* left\_element = *self*.left.pop();  
 *let mut* right\_element = *self*.right.pop();  
 *let mut* half\_segment\_length = (exp >> 1);  
 *while* (right\_element != *None* && left\_element >= right\_element && half\_segment\_length > 0) {  
 *match* left\_element.take() {  
 *Some*(value) => {  
 buffer.push(value);  
 }  
 *None* => {  
 *break*;  
 }  
 }  
 left\_element = *self*.left.pop();  
 half\_segment\_length -= 1;  
 }  
 *if* (left\_element != *None*) {  
 left\_element.take().map(|value| { *self*.left.push(value); });  
 }

*let mut* buffer\_element = buffer.pop();  
 half\_segment\_length = (exp >> 1);  
 *while* (buffer\_element != *None*) {  
 *if* (half\_segment\_length > 0 && right\_element != *None* && right\_element < buffer\_element) {  
 *match* right\_element.take() {  
 *Some*(value) => {  
 *self*.left.push(value);  
 }  
 *None* => {  
 *break*;  
 }  
 }  
 right\_element = *self*.right.pop();  
 half\_segment\_length -= 1;  
 } *else* {  
 *match* buffer\_element.take() {  
 *Some*(value) => {  
 *self*.left.push(value);  
 }  
 *None* => {  
 *break*;  
 }  
 }  
 buffer\_element = buffer.pop();  
 }  
 }  
 *if* (right\_element != *None*) {  
 right\_element.take().map(|value| { *self*.right.push(value); });  
 }  
 }  
 }  
 }  
 exp = exp << 1;  
 }  
 }  
}

It will be a pity, if we don’t implement the reverse function. The reverse function is very useful and very easy to implement:

*pub fn* reverse(&*mut self*) {  
 mem::swap(&*mut self*.left, &*mut self*.right);  
}

It is that easy, left side Singly-Linked-List and right side Singly-Linked-List values are swapped and Doubly-Linked-List is reversed!

We could be able to print the whole data; Therefore, we will create functions for printing:

*pub fn* print\_line(&*mut self*)  
 *where* T: std::fmt::*Debug*{  
 *self*.check\_empty();  
 *let* sz = *self*.size();  
 *for* x *in* 0..sz {  
 print!("{:?} ", *self*.get(x));  
 }  
 println!();  
}

It will be checked if the list is empty. Every element will be printed in one line. It’s required for type generic to implement std::fmt::*Debug* trait.

The same logic will be applied on the following function, but separator will be added:

*pub fn* print\_fmt(&*mut self*, separator: *char*)  
 *where* T: std::fmt::*Debug*{  
 *self*.check\_empty();  
 *let* sz = *self*.size();  
 *for* x *in* 0..sz {  
 print!("{:?}{}", *self*.get(x), separator);  
 }  
 *if* (separator != '\n') {  
 println!();  
 }  
}

The last thing we have to implement is the Drop trait. It will be implemented simply:

* Left side Singly-Linked-List will be dropped
* Right side Singly-Linked-List will be dropped
* Size of Doubly-Linked-List will become zero

*pub trait Drop* {  
 *fn* drop(&*mut self*);  
}  
  
*impl*<T> *Drop for* DoublyLinkedList<T> {  
 *fn* drop(&*mut self*) {  
 *self*.left.drop();  
 *self*.right.drop();  
 *self*.size = 0;  
 }  
}

We wrote all of the functionality of Doubly-Linked-List We can test some functions!

In main.rs we have to include a doubly linked list module and traits.

*use* std::fmt::*Debug*;  
  
*mod* doubly\_linked\_list;  
  
*use* doubly\_linked\_list::DoublyLinkedList;  
*use crate*::doubly\_linked\_list::*Drop*;

We will also create a struct which will be pushed into the Doubly-Linked-List:

#[derive(Debug, PartialEq, PartialOrd)]  
*pub struct* Person {  
 name: String,  
 age: *u8*,  
 height: *f32*,  
}

Let’s Create a new instance of Doubly-Linked-List and push Person types into it:

*fn* main() {  
 *let mut* dl: DoublyLinkedList<Person> = DoublyLinkedList::*new*();  
 dl.push\_back(Person {  
 name: String::*from*("John"),  
 age: 15,  
 height: 1.7,  
 });  
 dl.push\_back(Person {  
 name: String::*from*("John"),  
 age: 25,  
 height: 2.0,  
 });  
 dl.push\_back(Person {  
 name: String::*from*("John"),  
 age: 18,  
 height: 2.0,  
 });  
 dl.push(Person {  
 name: String::*from*("Emily"),  
 age: 12,  
 height: 1.25,  
 }, 3);

//tests will be written here  
}

Test 1:

let’s print out the first element of the Doubly-Linked-List:

println!("The first element: {:?}",dl.get(0));

output:

The first element: Person { name: "John", age: 15, height: 1.7 }

Test 2:

Print whole data in one Line:

println!("Initial order:");  
dl.print\_line();

output:

Initial order:

Person { name: "John", age: 15, height: 1.7 } Person { name: "John", age: 25, height: 2.0 } Person { name: "John", age: 18, height: 2.0 } Person { name: "Emily", age: 12, height: 1.25 }

Test 3:

Checking if data will be reversed correctly:

dl.reverse();  
println!("Reversed order:");  
dl.print\_line();

output:

Reversed order:

Person { name: "Emily", age: 12, height: 1.25 } Person { name: "John", age: 18, height: 2.0 } Person { name: "John", age: 25, height: 2.0 } Person { name: "John", age: 15, height: 1.7 }

Test 4:

Checking sort and print\_fmt functions:

dl.sort();  
println!("Sorted order:");  
dl.print\_fmt('\n');

Output:

Sorted order:

Person { name: "Emily", age: 12, height: 1.25 }

Person { name: "John", age: 15, height: 1.7 }

Person { name: "John", age: 18, height: 2.0 }

Person { name: "John", age: 25, height: 2.0 }

Test 5:

Checking pop\_back function:

dl.pop\_back();  
println!("after popping back:");  
dl.print\_fmt('\n');

Output:

after popping back:

Person { name: "Emily", age: 12, height: 1.25 }

Person { name: "John", age: 15, height: 1.7 }

Person { name: "John", age: 18, height: 2.0 }

Test 6:

Checking pop function:

dl.pop(1);  
println!("after popping 1st:");  
dl.print\_fmt('\n');

Output:

after popping 1st:

Person { name: "Emily", age: 12, height: 1.25 }

Person { name: "John", age: 18, height: 2.0 }

Test 7:

Checking drop function:

dl.drop();  
println!("{}",dl.size());

Output:

0

Since we checked the functionality, and it is working fine, efficient, fast, and safe. We can now say that: Doubly-Linked-List is implemented in Rust!!!