

```
1 use std::cell::RefCell;
2 use std::fmt::{Debug, Display, Formatter, Result as FmtResult};
3 use std::rc::Rc;
4
5 // Node struct
6 #[derive(Debug)]
7 struct Node<T> {
8     element: T,
9     prev: Option<Rc<RefCell<Node<T>>>>,
10    next: Option<Rc<RefCell<Node<T>>>>,
11 }
12
13 // Implement PartialEq for Node
14 impl<T: PartialEq> PartialEq for Node<T> {
15     fn eq(&self, other: &Self) -> bool {
16         self.element == other.element
17     }
18 }
19
20 // Implement Display for Node
21 impl<T: Display> Display for Node<T> {
22     fn fmt(&self, f: &mut Formatter<'_>) -> FmtResult {
23         write!(f, "{}", self.element)
24     }
25 }
26
27 // List struct
28 struct List<T> {
29     head: Option<Rc<RefCell<Node<T>>>>,
30     tail: Option<Rc<RefCell<Node<T>>>>,
31     size: usize,
32 }
33
34 impl<T: PartialEq + Debug + Display + Clone> List<T> {
35     // Create a new empty list
36     fn new() -> Self {
37         Self {
38
```

```

39         head: None,
40         tail: None,
41         size: 0,
42     }
43 }
44
45 // Print the elements of the list
46 fn print_list(&self) {
47     let mut current = self.head.clone();
48     while let Some(node) = current {
49         println!("{}", node.borrow().element);
50         current = node.borrow().next.clone();
51     }
52 }
53
54 // Add an element to the front of the list
55 fn push(&mut self, element: T) {
56     let new_node = Rc::new(RefCell::new(Node {
57         element,
58         prev: None,
59         next: self.head.clone(),
60     }));
61
62     if let Some(old_head) = self.head.clone() {
63         old_head.borrow_mut().prev = Some(new_node.clone());
64     } else {
65         self.tail = Some(new_node.clone());
66     }
67
68     self.head = Some(new_node);
69     self.size += 1;
70 }
71
72 // Remove and return the front element
73 fn pop(&mut self) -> Option<T> {
74     self.head.take().map(|old_head| {
75         self.head = old_head.borrow().next.clone();
76
77         if let Some(new_head) = self.head.clone() {
78             new_head.borrow_mut().prev = None;
79         } else {

```

```

80         self.tail = None;
81     }
82
83     self.size -= 1;
84     Rc::try_unwrap(old_head).ok().unwrap().into_inner().element
85 })
86 }
87
88 // Add an element to the back of the list
89 fn push_back(&mut self, element: T) {
90     let new_node = Rc::new(RefCell::new(Node {
91         element,
92         prev: self.tail.clone(),
93         next: None,
94     }));
95
96     if let Some(old_tail) = self.tail.clone() {
97         old_tail.borrow_mut().next = Some(new_node.clone());
98     } else {
99         self.head = Some(new_node.clone());
100     }
101
102     self.tail = Some(new_node);
103     self.size += 1;
104 }
105
106 // Remove and return the last element
107 fn pop_back(&mut self) -> Option<T> {
108     self.tail.take().map(|old_tail| {
109         self.tail = old_tail.borrow().prev.clone();
110
111         if let Some(new_tail) = self.tail.clone() {
112             new_tail.borrow_mut().next = None;
113         } else {
114             self.head = None;
115         }
116
117         self.size -= 1;
118         Rc::try_unwrap(old_tail).ok().unwrap().into_inner().element
119     })
120 }

```

```

121 }
122
123 // Implement PartialEq for List
124 impl<T: PartialEq + Clone> PartialEq for List<T> {
125     fn eq(&self, other: &Self) -> bool {
126         if self.size != other.size {
127             return false;
128         }
129
130         let mut self_current = self.head.clone();
131         let mut other_current = other.head.clone();
132
133         while let (Some(self_node), Some(other_node)) = (self_current, other_current) {
134             if self_node.borrow().element != other_node.borrow().element {
135                 return false;
136             }
137
138             self_current = self_node.borrow().next.clone();
139             other_current = other_node.borrow().next.clone();
140         }
141
142         true
143     }
144 }
145
146 // Implement Debug for List
147 impl<T: Debug> Debug for List<T> {
148     fn fmt(&self, f: &mut Formatter<'_>) -> FmtResult {
149         let mut elements = vec![];
150         let mut current = self.head.clone();
151
152         while let Some(node) = current {
153             elements.push(format!("{:?}", node.borrow().element));
154             current = node.borrow().next.clone();
155         }
156
157         write!(f, "List [{}]", elements.join(", "))
158     }
159 }
160
161 fn main() {

```

```
162     let mut list: List<i32> = List::new();
163
164     // Push elements to the front
165     list.push(1);
166     list.push(2);
167     list.push(3);
168
169     // Print the list
170     list.print_list(); // Output: 3 2 1
171
172     // Pop the front element
173     println!("Popped: {:?}", list.pop()); // Output: Popped: Some(3)
174
175     // Push elements to the back
176     list.push_back(4);
177     list.push_back(5);
178
179     // Print the list again
180     list.print_list(); // Output: 2 1 4 5
181
182     // Pop the back element
183     println!("Popped Back: {:?}", list.pop_back()); // Output: Popped Back: Some(5)
184
185     // Print the list
186     list.print_list(); // Output: 2 1 4
187 }
```