```rust
use std::cell::RefCell;
use std::fmt::{Debug, Display, Formatter, Result as FmtResult};
use std::rc::Rc;

// Node struct
#[derive(Debug)]
struct Node<T> {
    element: T,
    prev: Option<Rc<RefCell<Node<T>>>>,
    next: Option<Rc<RefCell<Node<T>>>>,
}

// Implement PartialEq for Node
impl<T: PartialEq> PartialEq for Node<T> {
    fn eq(&self, other: &Self) -> bool {
        self.element == other.element
    }
}

// Implement Display for Node
impl<T: Display> Display for Node<T> {
    fn fmt(&self, f: &mut Formatter<'_>) -> FmtResult {
        write!(f, "{}", self.element)
    }
}

// List struct
struct List<T> {
    head: Option<Rc<RefCell<Node<T>>>>,
    tail: Option<Rc<RefCell<Node<T>>>>,
    size: usize,
}

impl<T: PartialEq + Debug + Display + Clone> List<T> {
    // Create a new empty list
    fn new() -> Self {
        Self {

```

```
            head: None,
            tail: None,
            size: 0,
        }
    }

    // Print the elements of the list
    fn print_list(&self) {
        let mut current = self.head.clone();
        while let Some(node) = current {
            println!("{} ", node.borrow().element);
            current = node.borrow().next.clone();
        }
    }

    // Add an element to the front of the list
    fn push(&mut self, element: T) {
        let new_node = Rc::new(RefCell::new(Node {
            element,
            prev: None,
            next: self.head.clone(),
        }));

        if let Some(old_head) = self.head.clone() {
            old_head.borrow_mut().prev = Some(new_node.clone());
        } else {
            self.tail = Some(new_node.clone());
        }

        self.head = Some(new_node);
        self.size += 1;
    }

    // Remove and return the front element
    fn pop(&mut self) -> Option<T> {
        self.head.take().map(|old_head| {
            self.head = old_head.borrow().next.clone();

            if let Some(new_head) = self.head.clone() {
                new_head.borrow_mut().prev = None;
            } else {
```

```
                self.tail = None;
            }

            self.size -= 1;
            Rc::try_unwrap(old_head).ok().unwrap().into_inner().element
        })
    }

    // Add an element to the back of the list
    fn push_back(&mut self, element: T) {
        let new_node = Rc::new(RefCell::new(Node {
            element,
            prev: self.tail.clone(),
            next: None,
        }));

        if let Some(old_tail) = self.tail.clone() {
            old_tail.borrow_mut().next = Some(new_node.clone());
        } else {
            self.head = Some(new_node.clone());
        }

        self.tail = Some(new_node);
        self.size += 1;
    }

    // Remove and return the last element
    fn pop_back(&mut self) -> Option<T> {
        self.tail.take().map(|old_tail| {
            self.tail = old_tail.borrow().prev.clone();

            if let Some(new_tail) = self.tail.clone() {
                new_tail.borrow_mut().next = None;
            } else {
                self.head = None;
            }

            self.size -= 1;
            Rc::try_unwrap(old_tail).ok().unwrap().into_inner().element
        })
    }
```

```
}

// Implement PartialEq for List
impl<T: PartialEq + Clone> PartialEq for List<T> {
    fn eq(&self, other: &Self) -> bool {
        if self.size != other.size {
            return false;
        }

        let mut self_current = self.head.clone();
        let mut other_current = other.head.clone();

        while let (Some(self_node), Some(other_node)) = (self_current, other_current) {
            if self_node.borrow().element != other_node.borrow().element {
                return false;
            }

            self_current = self_node.borrow().next.clone();
            other_current = other_node.borrow().next.clone();
        }

        true
    }
}

// Implement Debug for List
impl<T: Debug> Debug for List<T> {
    fn fmt(&self, f: &mut Formatter<'_>) -> FmtResult {
        let mut elements = vec![];
        let mut current = self.head.clone();

        while let Some(node) = current {
            elements.push(format!("{:?}", node.borrow().element));
            current = node.borrow().next.clone();
        }

        write!(f, "List [{}]", elements.join(", "))
    }
}

fn main() {
```

```
    let mut list: List<i32> = List::new();

    // Push elements to the front
    list.push(1);
    list.push(2);
    list.push(3);

    // Print the list
    list.print_list(); // Output: 3 2 1

    // Pop the front element
    println!("Popped: {:?}", list.pop()); // Output: Popped: Some(3)

    // Push elements to the back
    list.push_back(4);
    list.push_back(5);

    // Print the list again
    list.print_list(); // Output: 2 1 4 5

    // Pop the back element
    println!("Popped Back: {:?}", list.pop_back()); // Output: Popped Back: Some(5)

    // Print the list
    list.print_list(); // Output: 2 1 4
}
```
