Table of Contents

List

The List package provides an immutable linked list data structure with a variety of utility functions for functional programming.

Table of Contents

Overview

List is a functional, immutable data structure that supports efficient traversal, transformation, and manipulation. It is particularly useful for recursive algorithms and scenarios where immutability is required.

Performance

- **prepend**: O(1)
- **length**: O(n)
- **map/filter**: O(n)
- **concatenate**: O(n)
- **reverse**: O(n)
- **nth**: O(n)
- **sort**: O(n log n)
- **flatten**: O(n * m), where m is the average inner list length
- **space complexity**: O(n)

Usage

Create

You can create an empty list or a list from an array.

```

test {
  let empty_list : @list.List[Int] = @list.new()
  assert_true(empty_list.is_empty())
  let list = @list.of([1, 2, 3, 4, 5])
  assert_eq(list, @list.of([1, 2, 3, 4, 5]))
}
```

Basic Operations

Prepend

Add an element to the beginning of the list.

```

test {
  let list = @list.of([2, 3, 4, 5]).prepend(1)
  assert_eq(list, @list.of([1, 2, 3, 4, 5]))
}
```

Length

Get the number of elements in the list.

```

test {
  let list = @list.of([1, 2, 3, 4, 5])
  assert_eq(list.length(), 5)
}
```

Check if Empty

Determine if the list is empty.

```

test {
  let empty_list : @list.List[Int] = @list.new()
  assert_eq(empty_list.is_empty(), true)
}
```

Access Elements

Head

Get the first element of the list as an Option.

```
test {
  let list = @list.of([1, 2, 3, 4, 5])
  assert_eq(list.head(), Some(1))
}
```

Tail

Get the list without its first element.

```
test {
  let list = @list.of([1, 2, 3, 4, 5])
  assert_eq(list.unsafe_tail(), @list.of([2, 3, 4, 5]))
}
```

Nth Element

Get the nth element of the list as an Option.

```
test {
  let list = @list.of([1, 2, 3, 4, 5])
  assert_eq(list.nth(2), Some(3))
}
```

Iteration

Each

Iterate over the elements of the list.

```
test {
  let arr = []
  @list.of([1, 2, 3, 4, 5]).each(x => arr.push(x))
  assert_eq(arr, [1, 2, 3, 4, 5])
}
```

Map

Transform each element of the list.

```
test {
  let list = @list.of([1, 2, 3, 4, 5]).map(x => x * 2)
  assert_eq(list, @list.of([2, 4, 6, 8, 10]))
}
```

Filter

Keep elements that satisfy a predicate.

```
test {
  let list = @list.of([1, 2, 3, 4, 5]).filter(x => x % 2 == 0)
  assert_eq(list, @list.of([2, 4]))
}
```

Advanced Operations

Reverse

Reverse the list.

```
test {
  let list = @list.of([1, 2, 3, 4, 5]).rev()
  assert_eq(list, @list.of([5, 4, 3, 2, 1]))
}
```

Concatenate

Concatenate two lists.

```
test {
  let list = @list.of([1, 2, 3]).concat(@list.of([4, 5]))
  assert_eq(list, @list.of([1, 2, 3, 4, 5]))
}
```

Flatten

Flatten a list of lists.

```
test {
  let list = @list.of([@list.of([1, 2]), @list.of([3, 4])]).flatten()
  assert_eq(list, @list.of([1, 2, 3, 4]))
}
```

Sort

Sort the list in ascending order.

```
test {
  let list = @list.of([3, 1, 4, 1, 5, 9]).sort()
  assert_eq(list, @list.of([1, 1, 3, 4, 5, 9]))
}
```

Conversion

To Array

Convert a list to an array.

```

test {
  let list = @list.of([1, 2, 3, 4, 5])
  assert_eq(list.to_array(), [1, 2, 3, 4, 5])
}
```

From Array

Create a list from an array.

```

test {
  let list = @list.from_array([1, 2, 3, 4, 5])
  assert_eq(list, @list.of([1, 2, 3, 4, 5]))
}
```

Equality

Lists with the same elements in the same order are considered equal.

```

test {
  let list1 = @list.of([1, 2, 3])
  let list2 = @list.of([1, 2, 3])
  assert_eq(list1 == list2, true)
}
```

Error Handling Best Practices

When accessing elements that might not exist, use pattern matching for safety:

```

fn safe_head(list : @list.List[Int]) -> Int {
  match list.head() {
    Some(value) => value
    None => 0
  }
}


test {
  let list = @list.of([1, 2, 3])
  assert_eq(safe_head(list), 1)
  let empty_list : @list.List[Int] = @list.new()
  assert_eq(safe_head(empty_list), 0)
}
```

Additional Error Cases

- **nth()** on an empty list or out-of-bounds index: Returns None.
- **tail()** on an empty list: Returns Empty.
- **sort()** with non-comparable elements: Throws a runtime error.

Implementation Notes

The List is implemented as a singly linked list. Operations like prepend and head are O(1), while operations like length and map are O(n).

Key properties of the implementation:

- Immutable by design
- Recursive-friendly
- Optimized for functional programming patterns

Comparison with Other Collections

- **@array.T**: Provides O(1) random access but is mutable; use when random access is required.
- **@list.T**: Immutable and optimized for recursive operations; use when immutability and functional patterns are required.

Choose List when you need:

- Immutable data structures
- Efficient prepend operations
- Functional programming patterns