Elixir Learning Materials

Linked Lists and Arrays

General information

- Wannes Fransen

Overview

- Comparison between
  - Arrays
  - Linked lists
- Purely functional implementation
  - Modifications are forbidden
  - Only creation of new objects is allowed

Memory Layout

Determining Length

Indexing

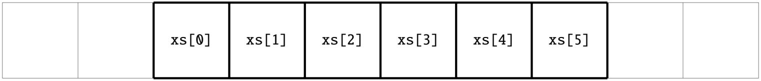
Updating

Adding to Front

Concatenation

Conclusion

Arrays



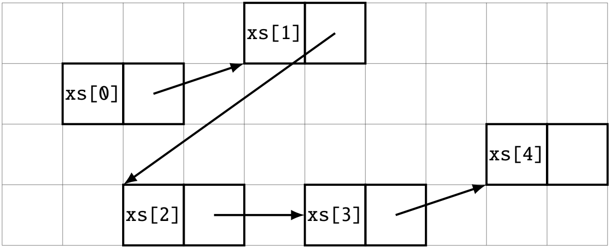
- One piece of contiguous memory

Arrays

```
template<typename T>
struct array
{
    // Where does it start?
    T* start;

    // For how long does it go on?
    int length;
};
```

Linked Lists



- List consists of series of nodes
- Each node has two fields
  - Item
  - Reference to next node
- Nodes scattered across memory

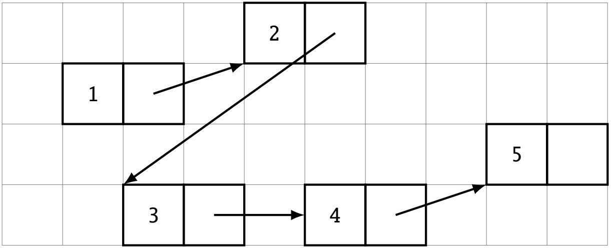
Linked Lists in Code

```
public class Node<T>
{
    public Node(T value, Node<T> next)
    {
        this.Value = value;
        this.Next = next;
    }

    public T Value { get; }

    public Node<T> Next { get; }
}
```

Creating a Linked List



```
var node5 = new Node<int>(5, null);
var node4 = new Node<int>(4, node5);
var node3 = new Node<int>(3, node4);
var node2 = new Node<int>(2, node3);
var node1 = new Node<int>(1, node2);
```

Memory Layout

Determining Length

Indexing

Updating

Adding to Front

Concatenation

Conclusion

Problem Statement

```
len([1,2,3,4,5,6,7])
```

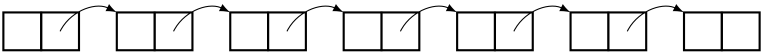
↓

```
7
```

Length of Array

- Array must keep track of length
- Computing length is immediate
- $O(1)$

Length of Linked List



Algorithm

- Follow nodes until we find `null`
- Count number of jumps necessary
- Takes longer for longer lists
- $O(n)$

Memory Layout

Determining Length

Indexing

Updating

Adding to Front

Concatenation

Conclusion

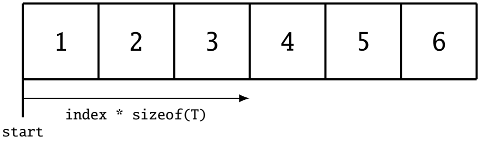
Problem Statement

```
[1, 2, 3, 4, 5, 6][3]
```

↓

```
4
```

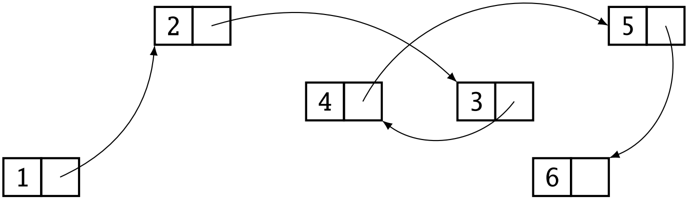
Indexing Array



Algorithm

- Memory location can be computed in a single step
- `location = start + index * sizeof(T)`
- Direct CPU support: only 1 instruction required
- Explains zero-indexing
- $O(1)$

Indexing Linked List



Algorithm

- Nodes are scattered unpredictably across memory
- Follow `Next` until `Next == null`
- Finding `n`th element takes `n` jumps
- $O(n)$

Memory Layout

Determining Length

Indexing

Updating

Adding to Front

Concatenation

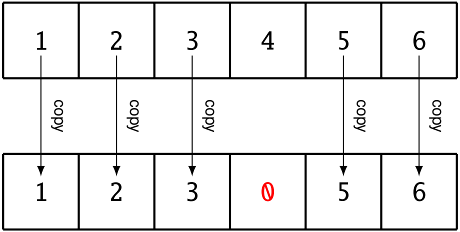
Conclusion

Problem Statement

```
[1, 2, 3, 4, 5, 6][3] = 0 ↓

[1, 2, 3, 0, 5, 6]
```

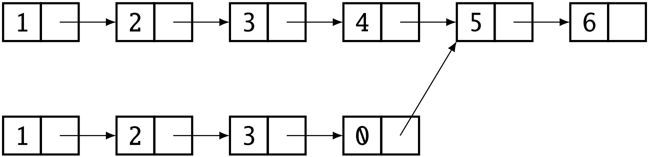
Updating an Array



Algorithm

- Requires copying entire array
- $O(n)$

Updating a Linked List



Algorithm

- Create new list
- Nodes after modified element can be reused
- $O(n)$

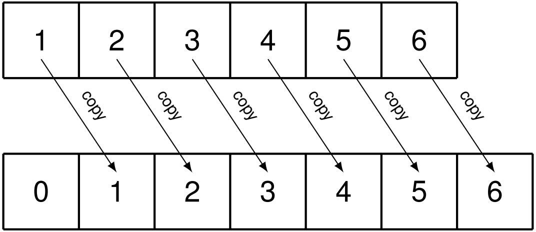
Problem Statement

```
prepend([1, 2, 3, 4, 5], 0)
```

↓

```
[0, 1, 2, 3, 4, 5]
```

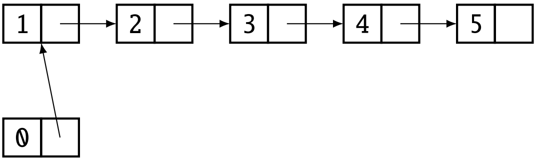
Add to Front of Array



Algorithm

- Create new array with larger size
- Copy all elements
- $O(n)$

Add to Front of Linked List



Algorithm

- Create new node
- Have it point to the (originally) first node
- $O(1)$

Memory Layout

Determining Length

Indexing

Updating

Adding to Front

Concatenation

Conclusion

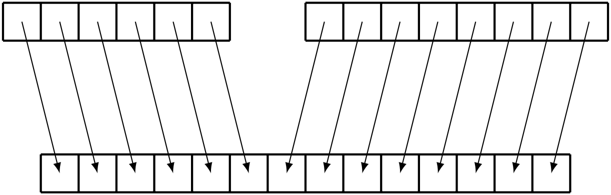
Problem Statement

```
[1,2,3,4,5] ++ [6,7,8,9,10,11,12,13]

↓

[1,2,3,4,5,6,7,8,9,10,11,12,13]
```

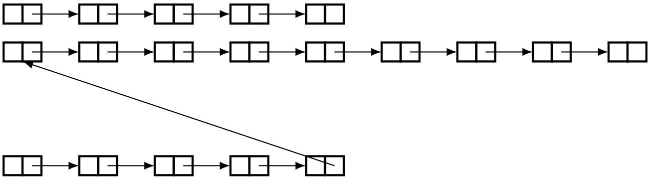
Updating an Array



Algorithm

- Requires both arries to be copied
- $O(n_1 + n_2)$

Updating a Linked List



Algorithm

- Only first list needs to be copied
- Second list can be safely reused
- Copy’s last node points to second list’s first node
- $O(n_1)$

Memory Layout

Determining Length

Indexing

Updating

Adding to Front

Concatenation

Conclusion

Comparison

	Array	**Linked List**
Length	$O(1)$	$O(n)$
Indexing	$O(1)$	$O(n)$
Updating	$O(n)$	$O(n)$
Add to front	$O(n)$	$O(1)$
Concatenation	$O(n_1 + n_2)$	$O(n_1)$

Usage

- Linked lists are often used for sequential processing
  - Move left to right
  - No indexing necessary
  - Build new list as you go
- Don’t treat linked lists as if they were arrays!