02393 Programming in C++ Module 10
Linked Lists

Sebastian Mödersheim

November 7, 2016

Lecture Plan

#	Date	Topic
1	29.8.	Introduction
2	5.9.	Basic C++
3	12.9.	Data Types, Pointers
4	19.9.	Data Types, Pointers / Libraries and Interfaces; Containers
5	26.9.	Libraries and Interfaces; Containers
6	3.10.	Classes and Objects I
7	10.10.	Classes and Objects II
		Efterårsferie
8	24.10.	Classes and Objects III
9	31.10.	Recursive Programming
10	7.11.	Lists
11	14.11.	Trees
12	21.11.	Novel C++ features
13	28.11.	Summary
	5.12.	Exam

Recursive Data Structures

Many data structures can be recursively defined:

- A natural number is 0 or a natural number plus one.
- A list is either empty, one item concatenated with a list.
  - ★ Typically we will only have homogenous lists where every list item has the same type (e.g. `int`).
- A tree is a node consisting of an item and a list of trees, called children. When a node has an empty list of children, we call it a leaf.

Recursive Data Structures

Many data structures can be recursively defined:

- A natural number is 0 or a natural number plus one.
  - ★ `data Nat = Zero | Succ Nat`
- A list is either empty, one item concatenated with a list.
  - ★ Typically we will only have homogenous lists where every list item has the same type (e.g. `int`).
  - ★ `data list` α `= Empty | Cons` α `(list` α`)`
- A tree is a node consisting of an item and a list of trees, called children. When a node has an empty list of children, we call it a leaf.
  - ★ `data tree` α `= Node` α `(list (tree` α`))`

Why Linked Lists?

- For many tasks, vectors (or arrays) are not so nice:
  - When insertion is a frequent operation
  - When we do not know the final size in advance
- Vector offers with `push_back` an amortisized-efficient solution for inserting at the end,
  - but inserting at any position other than the end necessarily requires to shift all following elements to the right!
- This can make programs inefficient and solutions harder to program.

Recursion to the Rescue!

Idea

- Do not store elements in sequence like an array.
- Rather, have a list where every element carries pointer to the next element.
- In this way, elements can be arbitrarily distributed over the memory (heap).
- The list can grow and shrink without ever moving around elements in memory.

Linked Lists

Recursive Definition

```
struct Node{
    int content;
    Node *next;
}
```

A Node has some content and points to a Node

A list can be then just an pointer to a `Node`:

- A `nullptr` would represent an empty list;
- Otherwise the pointer points to the first node in the list;

Linear Lists

Live Programming

- A `Queue` implemented as a linear list.
- Sorted insertion into a list.

Note: linked lists are provided by the STL. See
http://en.cppreference.com/w/cpp/container/list

Array vs. Lists

	Array	*List*
Iterative Access		
Random Access		
Insert/Delete		
Insert/Delete at end		

1

2

Array vs. Lists

	Array	*List*
Iterative Access	$O(1)$	$O(1)$
Random Access		
Insert/Delete		
Insert/Delete at end		

1

2

Array vs. Lists

	Array	*List*
Iterative Access	$O(1)$	$O(1)$
Random Access	$O(1)$	$O(n)$
Insert/Delete		
Insert/Delete at end		

1

2

Array vs. Lists

	Array	*List*
Iterative Access	$O(1)$	$O(1)$
Random Access	$O(1)$	$O(n)$
Insert/Delete	$O(n)$	$O(1)$[1]
Insert/Delete at end		

[1] given pointer to node before insertion/deletion

[2]

Array vs. Lists

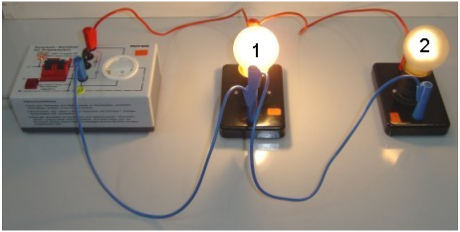
	Array	*List*
Iterative Access	$O(1)$	$O(1)$
Random Access	$O(1)$	$O(n)$
Insert/Delete	$O(n)$	$O(1)$[1]
Insert/Delete at end	$O(1)$[2]	$O(n)$

[1] given pointer to node before insertion/deletion

[2] amortized

Doubly-linked Lists

Some annoyances of single-linked lists: delete a pointed element, concatenate two lists, etc.
One possible solution: doubly-linked lists



Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	$O(N)$

Doubly-Linked Lists

Recursive Definition

```
struct Node{
    int content;
    Node *prev;
    Node *next;
}
```

A Node has some content and points to two Nodes: the previous one and the next one in the list.