02393 Programming in C++ Module 11
Recursive Data-Structures: Trees

Sebastian Mödersheim

October 14, 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

Vector vs. Lists

Recursive data-structures

```
struct listnode{
  int content; // for an integer list
  listnode *next;
};
```

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

1

2

Vector vs. Lists

Recursive data-structures

```
struct listnode{
  int content; // for an integer list
  listnode *next;
};
```

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

1

2

Vector vs. Lists

Recursive data-structures

```
struct listnode{
  int content; // for an integer list
  listnode *next;
};
```

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

1

2

Vector vs. Lists

Recursive data-structures

```
struct listnode{
  int content; // for an integer list
  listnode *next;
};
```

	Vector	*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]

Vector vs. Lists

Recursive data-structures

```
struct listnode{
  int content; // for an integer list
  listnode *next;
};
```

	Vector	*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]	$O(1)$

[1] given pointer to node before insertion/deletion

[2] amortized

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:
  - correct

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:
  - ★ correct
  - ★ well-structured

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:
  - ★ correct
  - ★ well-structured
  - ★ extendable/portable

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:
  - ★ correct
  - ★ well-structured
  - ★ extendable/portable
  - ★ readable and documented (comments!)

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:
  - ★ correct
  - ★ well-structured
  - ★ extendable/portable
  - ★ readable and documented (comments!)
- Don't sacrifice all this for a little optimization

Two Remarks on Efficiency

Efficiency is not everything

- Good code should be:
  - ★ correct
  - ★ well-structured
  - ★ extendable/portable
  - ★ readable and documented (comments!)
- Don't sacrifice all this for a little optimization

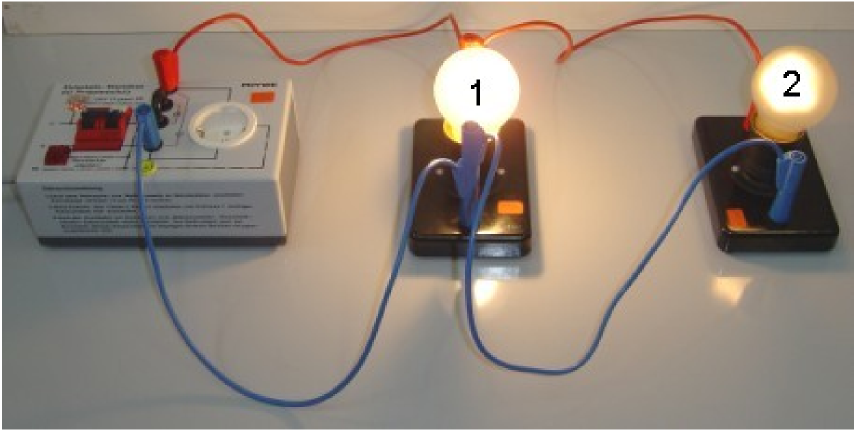
Think Big

- Don't focus on little optimizations.
- Think about $O(\cdot)$ runtime for large inputs.
- Do you expect such large inputs to happen in practice?
- Use profiling: which are the routines your program spends most time in?

Doubly-linked Lists

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

One possible solution: doubly-linked lists



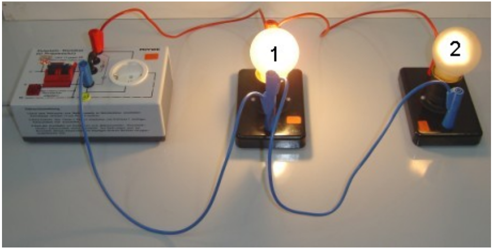
Doubly-Linked Lists

Recursive Definition

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

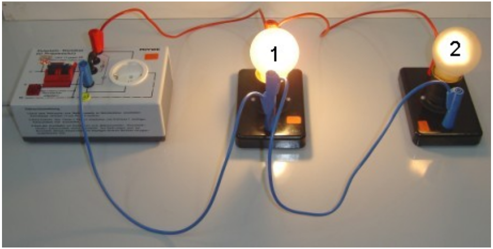
A Node has some content and points to two Nodes: the previous one and the next one in the list.

Double Linked Lists



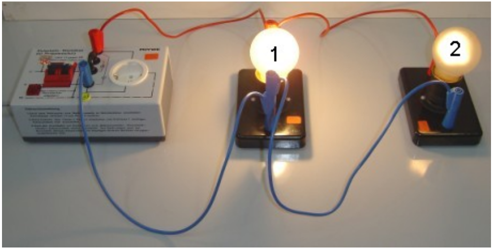
Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.			

Double Linked Lists



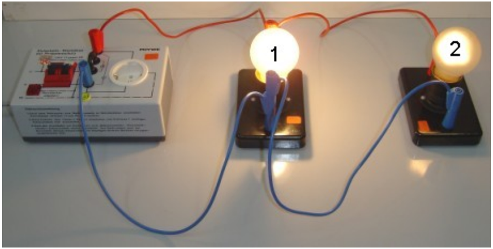
Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	

Double Linked Lists



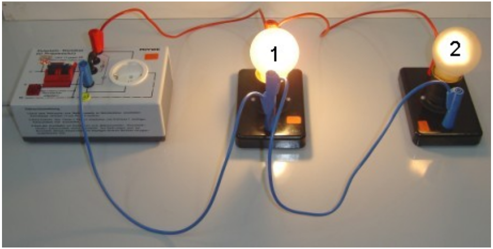
Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	$O(N)$

Double Linked Lists



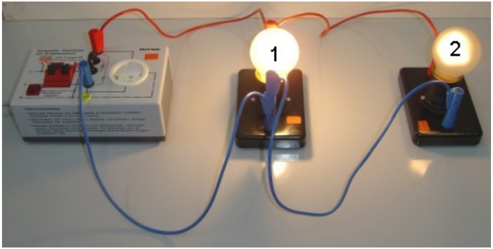
Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	$O(N)$
Special flag `isReversed`.			

Double Linked Lists



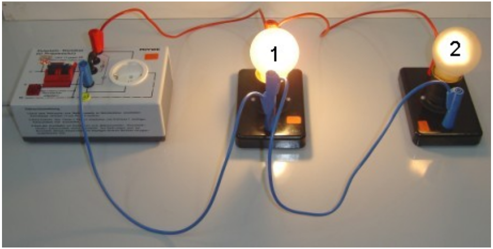
Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	$O(N)$
Special flag `isReversed`.	$O(1)$		

Double Linked Lists



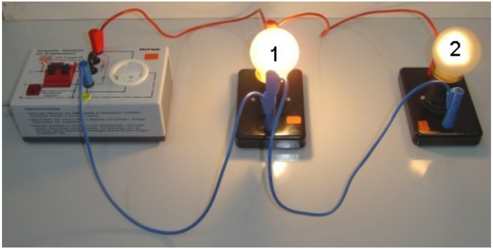
Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	$O(N)$
Special flag `isReversed`.	$O(1)$	$O(N)$	

Double Linked Lists



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

Double Linked Lists



Implementation	*Insert head*	*Concat*	*Reverse*
Concatenation by connecting the tail of one list with head of other list.	$O(1)$	$O(1)$	$O(N)$
Special flag `isReversed`.	$O(1)$	$O(N)$	$O(1)$
Possible?	$O(1)$	$O(1)$	$O(1)$

Recursive Data-Structures

Example: Binary tree of integers

Recursive Definition

```
struct Node{
 int content;
 Node *left;
 Node *right;
}
```

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)		

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array		

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list		

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$
Now: Binary Search Tree		

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$
Now: Binary Search Tree	$O(\log n)$	

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$
Now: Binary Search Tree	$O(\log n)$	$O(\log n)$
	if tree is balanced	

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$
Now: Binary Search Tree	$O(\log n)$	$O(\log n)$
	if tree is balanced	
Theoretical Optimum		

Review: The Map ADT

English-Danish Dictionary

red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$
Now: Binary Search Tree	$O(\log n)$	$O(\log n)$
	if tree is balanced	
Theoretical Optimum	$\Omega(\log n)$	

Review: The Map ADT

English-Danish Dictionary

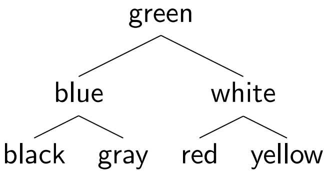
red	rød
green	grøn
blue	blå
yellow	gul
...	

What data structure to use?

Data Structure	*lookup*	*insert/delete*
unsorted array (Mod. 6)	$O(n)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$
sorted linked list	$O(n)$	$O(n)$
Now: Binary Search Tree	$O(\log n)$	$O(\log n)$
	if tree is balanced	
Theoretical Optimum	$\Omega(\log n)$	$\Omega(\log n)$

Map as a Tree

Organize the keys of a dictionary in a binary search tree:



- Binary search tree:
  - ★ Every node is larger than all nodes in the left subtree
  - ★ . . . and smaller than all nodes in the right subtree
- Mirrors what binary search does: split search space in half in each step of the space—if tree is balanced.
- The tree is dynamic: can easily grow and shrink (adding and removing entries).

Liveprogramming: A Binary Search Tree in C++

Reimplementation of the Map class from Module 6

Class of Trees

Another possible recursive definition of trees

```
class Tree{
public:
...
private:
  bool empty;
  int content;
  Tree *left;
  Tree *right;
}
```

Here a flag `empty` is used to denote empty trees. Every tree (node) will be a class. We will use this representation in the examples.

Class of Trees: Methods

Most methods we need to implement can be implemented using recursion.
Consider, for example, the size of a tree. A recursive formulation could be based on the idea that:

- the size of an empty tree is 0;
- the size of a non empty tree is 1 (for the root node) plus the size of its sub-trees

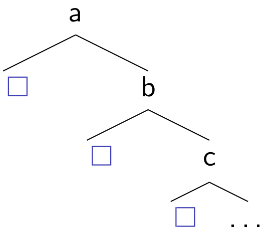
All other methods that we will see can exploit the recursive structure of trees.
Some of them are not easy to implement without recursion.

Balance

- Scenario: start with empty binary search tree, and insert nodes with increasing keys, e.g. $a, b, c, \ldots$

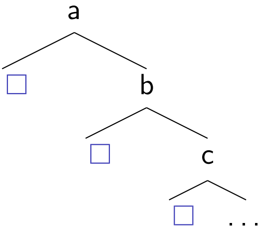
Balance

- Scenario: start with empty binary search tree, and insert nodes with increasing keys, e.g. $a, b, c, \ldots$
- The resulting tree is very unbalanced (□ written for nil-pointers):

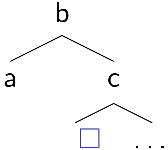


Balance

- Scenario: start with empty binary search tree, and insert nodes with increasing keys, e.g. $a, b, c, \ldots$
- The resulting tree is very unbalanced (□ written for nil-pointers):



- AVL trees: keep track of balance and perform re-arrangements:



Review: Sorting

Sorting Algorithms

	Time		Space
	average	worst-case	
Bubble sort			
Merge sort			
Quick sort			
...			
Theoretical Optimum			

Review: Sorting

Sorting Algorithms

	Time		Space
	average	worst-case	
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort			
Quick sort			
...			
Theoretical Optimum			

Review: Sorting

Sorting Algorithms

	Time		Space
	average	worst-case	
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort			
...			
Theoretical Optimum			

Review: Sorting

Sorting Algorithms

	Time		Space
	average	worst-case	
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort	$O(n \log n)$	$O(n^2)$	$O(1)$
...			
Theoretical Optimum			

Review: Sorting

Sorting Algorithms

	Time		Space
	average	worst-case	
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort	$O(n \log n)$	$O(n^2)$	$O(1)$
...			
Theoretical Optimum	$\Omega(n \log n)$	$\Omega(n \log n)$	$\Omega(1)$

Tree Sort

Tree sort algorithm:

- Successively insert all elements into a binary search tree.
- Print the tree in inorder

Tree Sort

Tree sort algorithm:

- Successively insert all elements into a binary search tree.
- Print the tree in inorder
- If the tree is balanced, it will have depth $O(\log n)$ for n elements and inserting one element costs $O(\log n)$ steps.

Tree Sort

Tree sort algorithm:

- Successively insert all elements into a binary search tree.
- Print the tree in inorder
- If the tree is balanced, it will have depth $O(\log n)$ for n elements and inserting one element costs $O(\log n)$ steps.
- Again $O(n \log n)$ for the entire sorting.