Summing up
[none of this will be on the exam :)]

Lab deadlines

You can submit lab 3 until Friday (even if it's a first submission)

If you've missed the final deadline for a lab, don't panic!

- On June the 3rd I will sit in my office all afternoon and you can show me your lab in person
- Let me know in advance so that I can reopen Fire for you

A catalogue of data structures

Basic data structures

Arrays: good for random access

- dynamic arrays: resizeable

Linked lists: good for sequential access

- many variants – doubly linked, etc.

Trees: good for hierarchical data

- special case: binary trees

Graphs: good for cyclic data

- many variants: weighted, directed, etc.

Some data structures are special

In machine language, the memory is an array of integers

- To the processor, everything is an array

In imperative languages, the memory is an *object graph* with references being edges

- To the imperative language, everything is a graph (or an array)

In functional languages, every algebraic data type is a kind of tree (cf. Lisp S-expressions)

- To the functional language, everything is a tree (or a function)

Everything is built from arrays, object graphs and algebraic data types

Basic ADTs

Maps: maintain a key/value relationship

- An array is a sort of map where the keys are array indices

Sets: like a map but with only keys, no values

Queue: add to one end, remove from the other

Stack: add and remove from the same end

Deque: add and remove from either end

Priority queue: add, remove minimum

Implementing maps and sets

A binary search tree

- Good performance if you can keep it balanced
- Has good random *and* sequential access: the best of both worlds

A hash table

- Very fast if you choose a good hash function

A linked list??

- ...pretty bad
- but used in the "chains" in a hash table

Implementing queues and deques

A linked list

- Can even get away with a singly-linked list

A circular array

A pair of lists (in a functional language)

- One for each end of the queue

Implementing stacks

Easier than queues:

- A linked list
- A dynamic array

Implementing priority queues

A binary heap

(More later...)

What we have studied

The data structures and ADTs above

+ algorithms that work on these data structures (sorting, Dijkstra's, etc.)

+ complexity

What we haven't had time for

Many algorithms

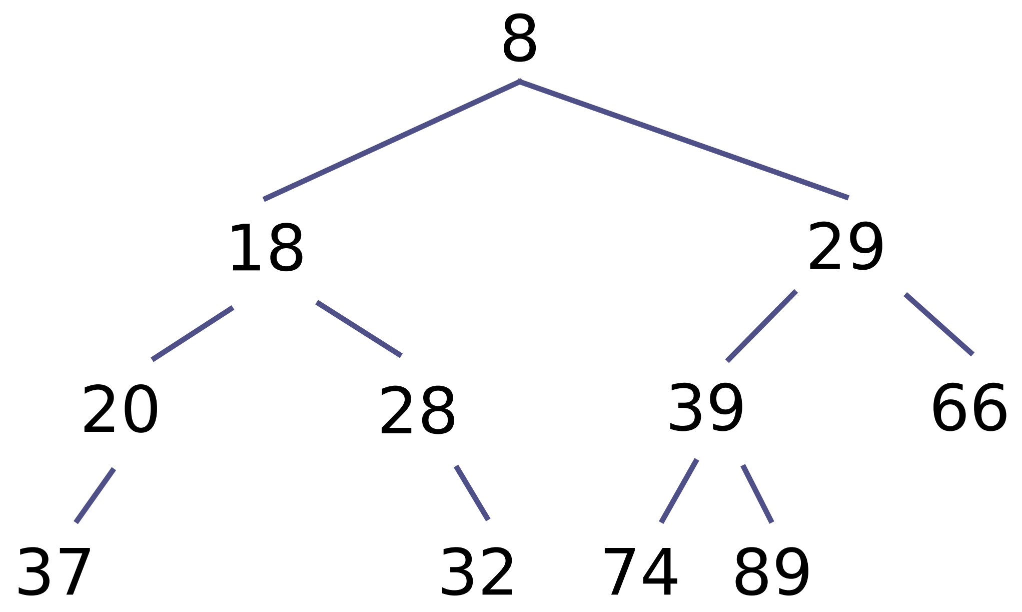
We have mostly concentrated on data structures, not algorithms

- though they go together

The course DIT600 talks much more about how to design algorithms (and leads to various advanced algorithms courses)

Merging priority queues

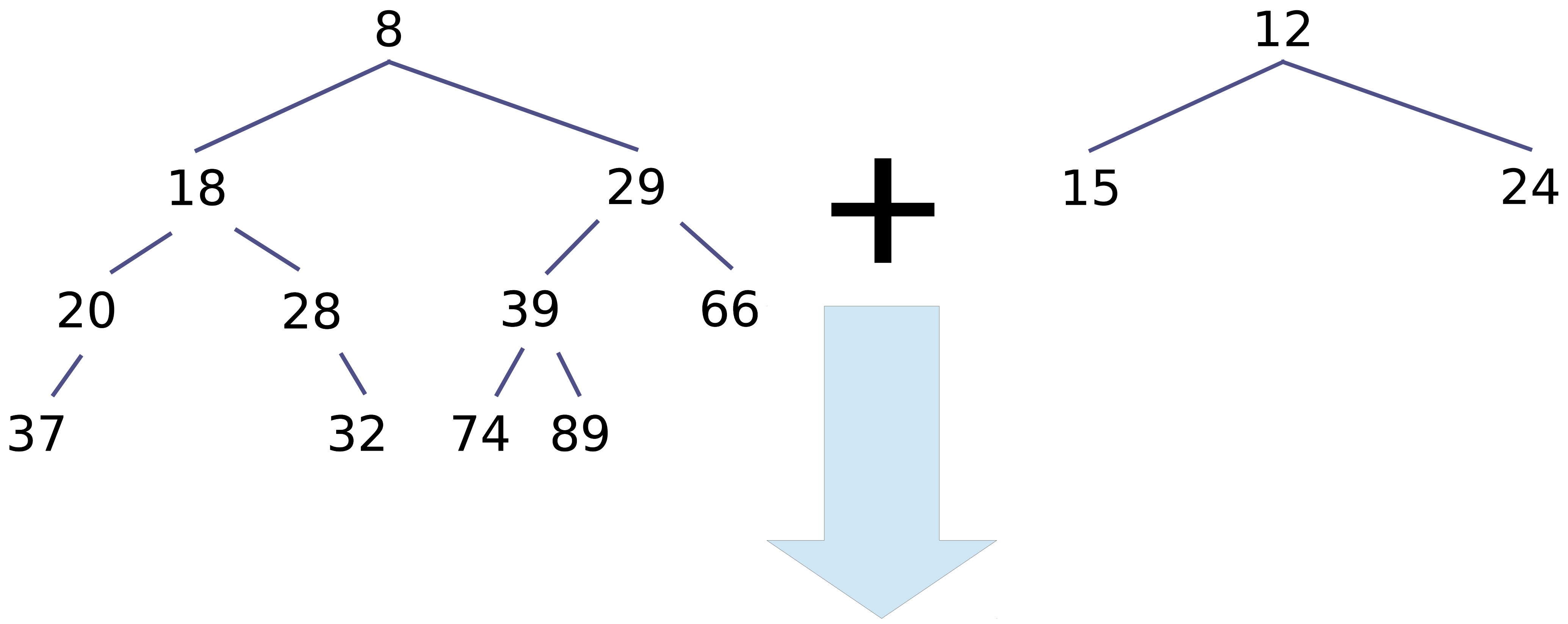
Go back to a binary heap as a binary tree satisfying the *heap property*



We encoded this tree as an array – but let's keep it as a binary tree instead

Merging priority queues

A merging priority queue supports *merging* two heaps into one:



a heap containing
8, 12, 15, 18, 20, 24, 28, 29, 32, 37, 39, 66, 74, 89

Merging priority queues

Insertion and deleting the minimum can both be implemented using merging!

To *add* an element to a heap:

- build a new heap containing just that one element
- merge the two heaps together!

To delete the minimum element:

- Well, the minimum element is at the root – so we want a heap containing everything except the root
- so merge the left and right children of the root!

Merging priority queues

By representing a heap as a binary tree with the heap property:

- We can implement a merge operation
- We can use this to implement add and deleteMin

We get a priority queue that's simpler than a binary heap – and supports an extra operation too!

Leftist heaps and *skew heaps* implement this idea

More exotic: Fibonacci heaps (very odd)

Amortised complexity

Think of dynamic arrays:

- adding an element normally takes O(1) time…
- …but occasionally it can take O(n) time
- The O(n) case happens seldom enough that adding *n* elements to an empty array takes O(n) time

We say that adding an element takes *amortised* O(1) time

Amortised analysis deals with data structures where:

- each operation is normally fast
- occasionally it can be slower
- there are so few slow operations that the fast operations "balance them out" and, in any sequence of operations, the *average time* per operation is still fast

Amortised complexity

Splay trees are a balanced BST having *amortised* O(log n) complexity

- The tree sometimes becomes unbalanced but this happens rarely enough that the average time per operation is still O(log n)

Skew heaps are a priority queue having amortised O(log n) merge

- Similar to leftist heaps but much simpler, and faster in practice!

Amortised complexity

By giving up *absolute* complexity bounds and going for amortised complexity instead, we can get data structures that are

- simpler
- often faster
- but harder to analyse the complexity of

See book chapters 22 (splay trees) and 23 (skew heaps)

Probabilistic algorithms

Sometimes it helps to make *random choices*

- Example: quicksort with a random pivot has *expected* O(n log n) complexity

Probabilistic algorithms and data structures use randomness in their implementation

- Downside: harder to analyse, small chance of poor performance (but if the probability is low enough…)

Skip lists: a nice map-like data structure with O(log n) expected complexity

Randomised splay tree: a balanced BST with O(log n) expected complexity

Functional data structures

Zippers: allow you to update functional data structures efficiently

- http://www.haskell.org/haskellwiki/Zipper

Finger trees: a sequence data type with almost magical complexity (O(1) access near each end of the sequence, O(log n) random access, O(log n) concatenation and splitting)

- http://www.soi.city.ac.uk/~ross/papers/FingerTree.pdf

Program specification

We wrote down invariants for our data structures that explain how they work and let us find bugs

By writing the invariant for a BST, we can check that BST insertion leaves us with a valid BST

- but we don't know that insertion actually inserts the new element, or that it doesn't remove existing elements

Program specification

We can *specify* the insertion function:

- define a function giving the set of all elements in a BST
- write a *postcondition* for insert, saying that the new BST contains all elements it did before, plus the new element

We can use these specifications as assertions to help find bugs (design by contract), to help with testing, or to prove that our programs are correct

- Course: Testing, Debugging and Verification (DIT082)

How to design a data structure

How to design a data structure

Here is the tempting approach:

- Just write down some datatype and hack something together that seems to work

Please don't do this!

Step 1

Write down what operations you need:

- add a *thing*, find a thing that has *this* property, …

Maybe there is already a data structure that does what you want!

If so, use it!

If not, proceed to step 2.

Step 2

Do you need a fancy data structure at all?

- There is no point designing a data structure with O(log n) everything, if in reality *n = 10*

"The First and Second Rules of Program Optimisation:

1. Don't do it.

2. (For experts only!): Don't do it yet."

See if you can get away with something simple, like a list, map or array

- and *profile your program* to see where the bottlenecks are

Step 3

So you need a fancy data structure, and you can't use an existing one

Can you adapt one?

- Example from the labs: adding decreaseKey() to a binary heap, by combining it with a map
- Example from the book (19.2): adding array functionality to a binary search tree

Step 4

Choose how you want to represent your data structure

- Is it an array, a list, a tree, a graph?

Make sure you understand the *meaning* of the representation

- Example: for a queue implemented as a circular array, the representation is an array and two integers *low* and *high*. The meaning (as a queue) of such an array is all elements between *low* and *high*.

Step 5

Choose an invariant for the data structure, if there is one

- Example: heap invariant, BST invariant

The invariant should normally guide the *searching* operations of your data structure

- An invariant might make it harder to *update* the data structure, so don't make it needlessly strong

This takes creativity!

Step 6

Finally, implement the operations, making sure they respect the invariant

- A good invariant will often *drive* the implementation: there will only be one sensible way to implement the operations

Use assertions to check the invariant at strategic places

Often, your idea will turn out to be wrong! Refine it and try again. It takes practice!

Designing a data structure

Picking a representation and an invariant takes creativity!

The only way to get better at it is to practise

The good news is, once you've picked an invariant, there's often only one sensible way to implement the operations

Let's see a few examples of how common data structures “might have been designed”

Example: queues as circular arrays

Attempt 1: implement a queue using a dynamic array

- an array plus a “high” index
- the queue contains all elements from 0 to high-1

Problem: removing an element from the front of the queue will take O(n) time

Attempt 2: add a “low” index, increment it to remove an element

- the queue contains all elements from low to high-1

Problem: what happens when “low” reaches the end of the array?

Solution: use a circular array and wrap around

- the queue contains all elements from low to high-1, possibly wrapping around

Example: binary heaps

We pick a binary tree as the representation. But finding the minimum element in a binary tree takes O(n) time

Answer: put the smallest element at the root

- with binary trees, it's a good idea to make the invariant hold for all subtrees too – the minimum of each subtree should be at the subtree's root
- this is then the heap invariant!

Example: binary heaps

Problem: we would really like to use an array for compactness

Solution: *represent* the tree by an array. This only works for complete binary trees – so add completeness to the invariant

- to implement add: add the element to the end of the array and fix the invariant
- only efficient way to delete an element: copy the final element over that one and reduce the size by 1 – then fix the invariant

So choosing the representation and invariant *forces* you to implement add and delete a particular way

Example: binary search trees

We pick a binary tree as the representation. But finding the minimum element in a binary tree takes O(n) time

Answer: add the BST invariant – then we can do lookups in O(log n) time on balanced trees

- there is only one natural way to implement insert and delete that preserves the invariant

How to keep the tree balanced? Some kind of balance invariant (e.g. AVL, red-black)

The invariant should be weak enough that we can maintain it, but strong enough to enforce balance

Example: 2-3 trees

Problem: balanced binary search trees are hard because they need to maintain weird invariants

- the invariant “all leaves have the same depth” does not work because the number of nodes in such a tree must be a power of two minus one

Answer: be more lenient, so that a node can even have 3 children

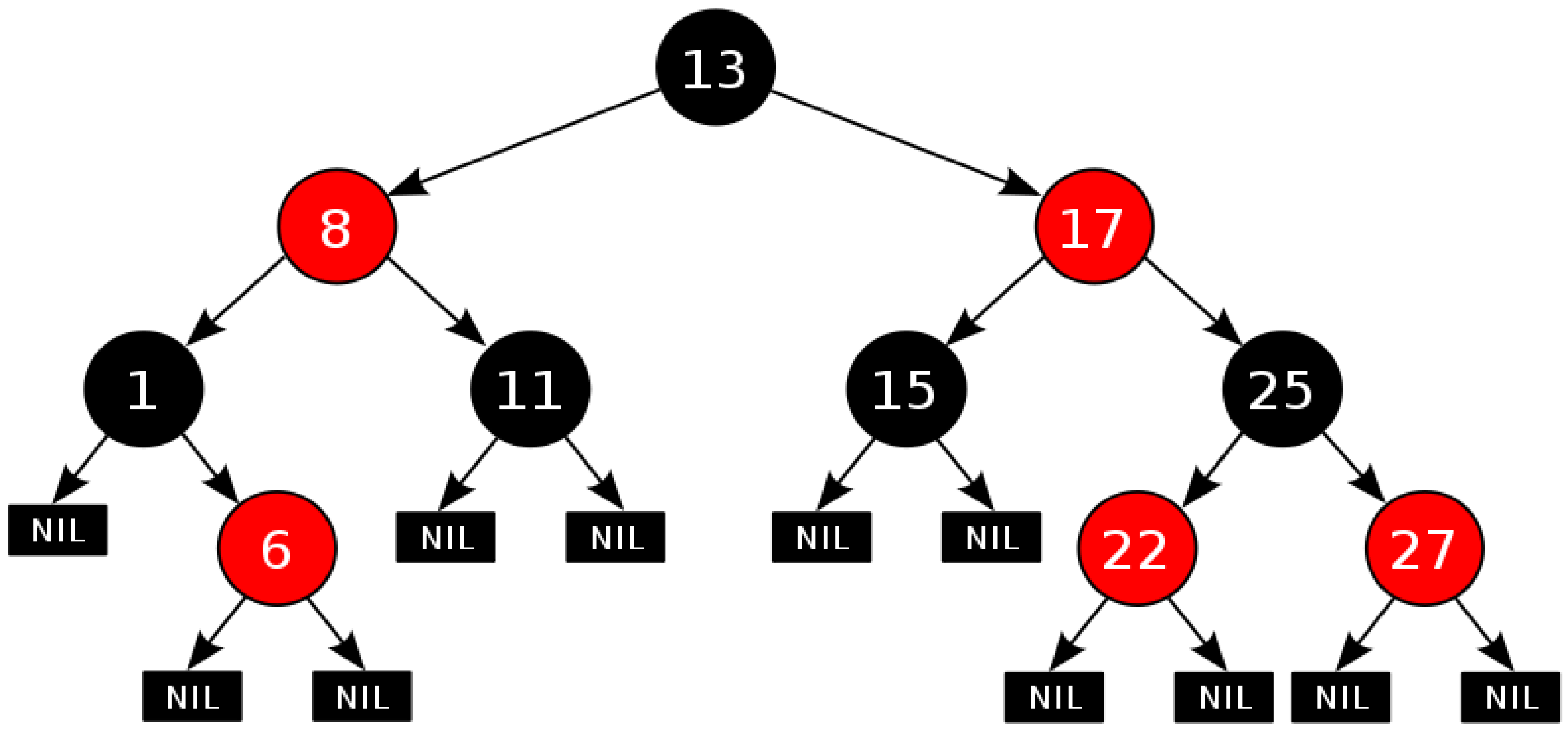
- now the invariant “all leaves have the same depth” works, and life is much simpler!

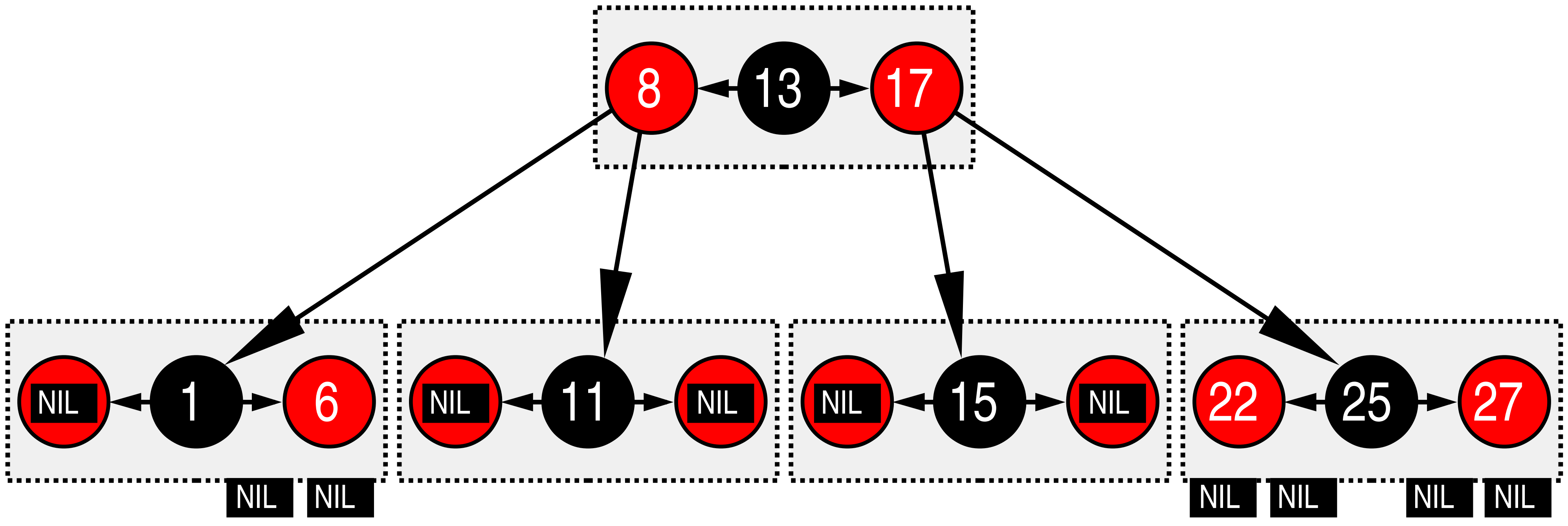
Example: red-black trees

Binary search trees with a *weird* balance invariant. How can you come up with that?

Answer: don't! Instead, remember that a red-black tree is a clever encoding of a 2-3-4 tree.

A red-black tree is a 2-3-4 tree





Example: red-black trees

Problem: 2-3-4 trees are a pain to implement because of the many different cases with 2-nodes, 3-nodes and 4-nodes

Answer: represent a 2-3-4 tree using a binary tree!

- a 3-node must be encoded by two nodes, a parent and a child
- a 4-node must be encoded by three nodes, a parent and two children

How will we tell if a part of the tree represents a 2-node, a 3-node or a 4-node?

- colour all the nodes that represent a 2-node, or the “start” of a 3-node or 4-node, black
- colour all the nodes that represent a “part” of a 3-node or 4-node red

Now just translate the 2-3-4 tree operations to this new representation, and you have a red-black tree!

How to get better at this creative step?

Study other people's ideas!

- http://en.wikipedia.org/wiki/List_of_data_structures
- Book: Programming Pearls **(excellent book)**
- Book: Purely Functional Data Structures
- The documentation for Haskell data structures often has a link to a paper explaining the ideas – looking at the source code also helps
- Download the Java source code at http://download.java.net/openjdk/jdk7/and look at how things are implemented

Design some data structures!

- Just try things, even if it doesn't work

The exam

30th of May, 14:00 – 18:00, VV

The exam

You can bring a fusklapp, written (or printed) on both sides

But *no textbook*!

Two sections: 6 normal questions and 3 harder questions

- Answer 4 out of 6 normal questions (plus pass the labs) to get a G
- Answer 4 out of 6 normal questions, plus 2 out of 3 harder questions, to get a VG

The exam

Same style as last year's exam – so look at that!

Best exam preparation: try last year's exam, do the exercises, make sure you understand the labs

What you need to know: the following!

Data structures

Arrays, dynamic arrays

Linked lists (single-linked, doubly-linked, header nodes, circular, etc.)

Binary trees, binary search trees, AVL trees, red-black trees, 2-3 trees

- not deletion for AVL, red-black or 2-3 trees – but still for plain BSTs!

Hash tables

- Rehashing, linear probing, linear chaining – not how to construct a good hash function

Graphs (weighted, unweighted, directed, undirected), adjacency lists, adjacency matrices

Binary heaps

ADTs and their implementation

The basics of the Java collections framework (iterators etc.)

Maps and sets, implemented using BSTs and hash tables

Queues and deques, implemented using linked lists or circular arrays

Stacks, implemented using linked lists or dynamic arrays

Priority queues, implemented using binary heaps

Algorithms

Binary search

Algorithms on data structures (e.g., list insertion)

Tree traversal: in-order, pre-order, post-order

Graph algorithms:

- breadth-first and depth-first search
- Dijkstra's and Prim's algorithms (using a priority queue)

Sorting algorithms

Bubblesort, selection sort, insertion sort

- In-place versions

Quicksort, mergesort

- Strategies for choosing the pivot – first element, middle element, median-of-three, randomised

Heapsort

- In-place version

Counting sort

Theory

Complexity and big-O notation

- For iterative and recursive functions – basically, what's in the complexity hand-in

Data structure invariants

Tail recursion and how to eliminate it (in simple cases)

Good luck!