

# Week 4

## Binary Trees

- 1 Introduction
- 2 Trees
- 3 Implementing rooted trees
- 4 The notion of “binary search tree”
- 5 Queries in binary search trees
- 6 Insertion and deletion
- 7 Final remarks

This week we consider Part III “Data Structures” from CLRS:

- 1 the introduction to Part III on using sets on a computer
- 2 Section 10.4 on implementing rooted trees
- 3 Sections 12.1, 12.2, 12.3 on binary search trees.

The following week than we conclude the consideration of data structures by investigating Part V.

The most fundamental mathematical notion is that of a **set**:

- We have the possibility to determine the elements of a set.
- And we can form sets, either by some set-defining property, or by using already given sets (e.g., unions, intersections, differences).

Now to bring the eternal and infinite world of mathematics to a computer, we need to take care of

- construction of “objects”
- destruction of “objects”
- naming (basically of functions).

For this CLRS uses the (generic) ADT of “dynamic sets”.

# Dynamic sets: elements and keys

A “dynamic set” might contain

- pointers (or iterators) to objects,
- or the objects themselves (in Java this can be only integers and other primitive types, in C++ this is possible for every type of object).

Whatever the objects in a set are, access to them (especially for changing them) is only possible via a pointer (or iterator).

CLRS assumes that we always use some **key** to identify an object:

If the object is for example a record of personal attributes, then the name or some form of ID can be used as a key.

## Dynamic sets: elementship and search

With sets  $S$  we can “ask” whether “ $x \in S$ ” is true. This is the most basic set operation, and the equivalent for dynamic sets is the operation

- $\text{SEARCH}(S, k)$  for key  $k$  and dynamic set  $S$ , returning either a pointer (iterator) to an object in  $S$  with key  $k$ , or  $\text{NIL}$  if there is no such object.

We require the ability to extract the key from an object in  $S$ , and to compare keys for equality.

- 1 Storing  $S$  via an array (or a list),  $\text{SEARCH}$  can be performed in  $O(|S|)$  time (that is, *linear time*) by simple sequential search.
- 2 To do faster than this, typically in time  $O(\log(|S|))$  (*logarithmic time*), under various circumstances, is a major aim of data structures for dynamic sets.

# Dynamic sets: modifying operations

With the following operations we can build and change dynamic sets:

- $\text{INSERT}(S, x)$  inserts an object into dynamic set  $S$ , where  $x$  is either a pointer or the object itself.
- $\text{DELETE}(S, x)$  deletes an object from dynamic set  $S$ , where here  $x$  is a pointer (iterator) into  $S$ .

Note that the "x" in  $\text{DELETE}$  is of a different nature than the "x" in  $\text{INSERT}$ : It is a pointer (iterator!) *into* the (dynamic) set (and thus these two  $x$  are of different type).

The most important application is for creating a dynamic set: To create a set  $S$  of size  $n$ , call  $\text{INSERT}(S, -)$   $n$ -times.

# Dynamic sets: using order

Often it is assumed that a *linear order* is given on the keys:

- So besides “ $k == k'$  ?”
- we now can ask “ $k \leq k'$  ?”.

In practice using strict orders “ $<$ ” is more common, however this creates some (necessary) technical complications, which in this module we won't be much concerned about (we discuss issues when the need arises).

(These complications have to do with the notion of “equality” — related to this is that in Java (lacking operator overloading) you have two operations for checking “equality”: “ $==$ ” and “`.equals()`”.)

## Dynamic sets: four further operations

Given a linear order on the objects (to be put into the set), we have the following additional operations:

- $\text{MINIMUM}(S)$  returns a pointer (iterator) to the element with the smallest key
- $\text{MAXIMUM}(S)$  returns a pointer (iterator) to the element with the largest key
- $\text{SUCCESSOR}(S, x)$ , where  $x$  is a pointer (iterator) into  $S$ , returns the next element in  $S$  w.r.t. the order on the keys
- $\text{PREDECESSOR}(S, x)$ , where  $x$  is a pointer (iterator) into  $S$ , returns the previous element in  $S$ .

Operations for computing successors and predecessors can fail (there is no successor resp. predecessor iff we are already at the end resp. beginning), and in such cases we return NIL.

## Using sorting algorithms: static case

Dynamic sets can be realised using sorting, where we have to assume a linear order on the keys.

If the set  $S$  with  $n$  elements is to be built only once, at the beginning, from a sequence of elements, then storing the elements in an array and sorting them, using for example MERGE-SORT with time complexity  $O(n \cdot \log n)$ , is a good option:

- SEARCH then takes time  $O(\log n)$  (using binary search)
- while each of MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR takes constant time.

However we are concerned here with the *dynamic* case, where insertions and deletions are used in unpredictable ways. If we not assume that building the set is done (once and for all) at the beginning, but we want to have insertion and deletion, then the case is much less favourable.

## Using sorting algorithms: dynamic case

- Now INSERTION and DELETION take time  $O(\log(n) + n) = O(n)$ , searching first for the right insertion/deletion place, and then shifting the other elements appropriately,
- while the five other (non-modifying) operations still take logarithmic resp. constant time.

For practical applications, the linear complexity of insertion and deletion is not acceptable. And we also see that most of the intelligence of sophisticated searching algorithms is blown out of the window, and only some form of INSERTION-SORT (in combination with binary search) survived.

It could be said that data structures for dynamic sets try to introduce some of the intelligent methods into the dynamic framework, making insertion and deletion more efficient (necessarily at the cost of making the other operations (somewhat) less efficient).

Often not all of the operations for dynamic sets are needed, opening up the possibilities of specialised and more efficient implementations. Two important cases are as follows:

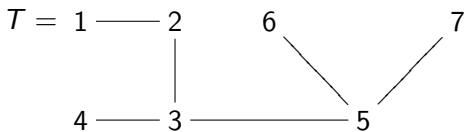
**dictionary** only INSERTION, DELETION and SEARCH are needed;

**priority queue** only INSERTION and MINIMUM resp. MAXIMUM (for min- resp. max-priority queues) are required.

# The notion of a “tree”: origins

(See Sections B.5.1, B.5.2 and B.5.3 in CLRS for more information on this topic.)

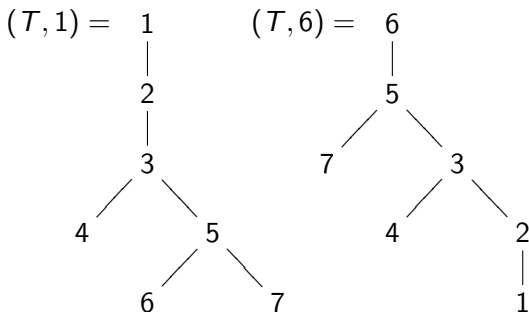
The original notion of a *tree* (in our context) comes from mathematics, where a tree is a “connected undirected graph without cycles”, e.g.



We have “vertices”  $1, \dots, 7$  and “edges” between them. From any vertices we can reach every other vertex, and this essentially in a unique way (this is the special property of trees).

## The notion of a "tree": roots

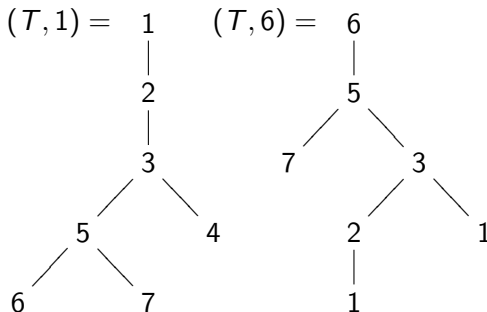
To obtain a *rooted tree*, one of the vertices of the tree has to be distinguished as a "root", and then the tree is typically drawn growing downwards (in the direction of reading). For our example  $T$  we can make seven rooted trees out of  $T$ , for example choosing vertex 1 resp. vertex 6 as the root:



The *leaves* of  $(T, 1)$  are 4, 6, 7, the leaves of  $(T, 6)$  are 7, 4, 1.

# The notion of a "tree": order

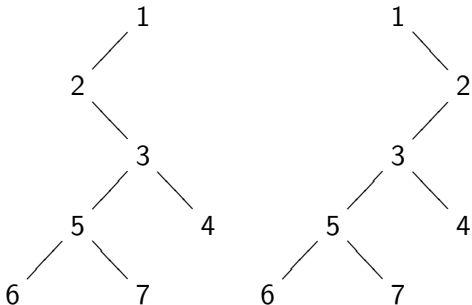
The *same* two rooted trees as before can be drawn differently, since there is no order on the children of a node, for example:



However, if we consider *ordered rooted trees*, then the order of the children of a node is part of the tree, and the above re-drawings are different *ordered* rooted trees.

# The notion of a “tree”: binary trees

To begin with, a *binary tree* is a rooted ordered tree, where every node which is not a leaf has either one or two children. But that's not quite complete: In case a node has one child, then for this child it must be said whether it is a *left child* or a *right child*. For example, the ordered rooted tree  $(T, 1)$  as given on the previous slide has  $2 \cdot 2 = 4$  different versions as binary tree, e.g.



# Nodes and pointers

A node of a binary tree has the following information naturally associated with it:

- its parent node
- its left and its right child
- its label.

There are four special cases:

- 1 the root has no parent node
- 2 a node may have only one child, left or right
- 3 a leaf has no children.

The links to parents and children can be represented by *pointers* (possibly the null pointer), while the label is just an “attribute” (a “data member” or “instance variable” of the class).

## Representing the example

Using as label just the key, we arrive at a class with four data members:

- "p" is the pointer to the parent
- "left" and "right" are pointers to left resp. right child
- "key" is the key.

Considering the first binary tree from two slides before, we can represent the tree by 7 nodes  $v_1, \dots, v_7$ , where each node is a pointer and where the data members are given in order p, left, right, key:

$$\begin{aligned}v_1^* &= (\text{NIL}, v_2, \text{NIL}, 1), v_2^* = (v_1, \text{NIL}, v_3, 2), v_3^* = (v_2, v_5, v_4, 3), \\v_4^* &= (v_3, \text{NIL}, \text{NIL}, 4), v_5^* = (v_3, v_6, v_7, 5), \\v_6 &= (v_5, \text{NIL}, \text{NIL}, 6), v_7 = (v_5, \text{NIL}, \text{NIL}, 7).\end{aligned}$$

# Remark on pointers

- Note that dereferenciation of pointers  $v$  is denoted by “ $v^*$ ” (the object to which it points).
- Since Java is “purely pointer-based”, one can not get at the address of an object, or to the object of a pointer: Every variable is a pointer, and using it in most cases(!) means to dereference it (an exception is for example when using  $==$ ).
- Another exception in Java is the handling of primitive types, where we can *not* have pointers.
- C and C++ have both *values* (objects) and *pointers*, and we have full symmetry (and full control):
  - For an object  $x$  the address is “ $\&x$ ”.
  - For a pointer  $p$  the object is “ $p^*$ ”.

# Rooted trees with unbounded branching

A few remarks in case there are more than two children:

- For  $k$ -ary trees (generalised binary trees) we just need to add further data members to the class, with appropriate names.
- This works best for small  $k$ . For large  $k$  or unbounded  $k$  (not known in advance) an easy way is to store the children in a list.
- Section 10.4 of CLRS contains a variation, where each element of the (singly-linked) list of children also contains a pointer to its parent.

With that we conclude our treatment of Chapter 10 from CLRS. Especially Sections 10.1 and 10.2 contain material used very often with algorithms and their implementations, so they are worth reading.

# The height of rooted trees

Given a rooted tree  $T$ , its **height**  $\text{ht}(T)$

- is the length of the longest path from the root to some leaf,
- where the length of a path is the number of edges on it.

The recursive definition is as follows:

- the trivial tree, just consisting of the root, has height 0;
- if a rooted tree  $T$  has subtrees  $T_1, \dots, T_m$  for  $m \geq 1$  (always at the root), then

$$\text{ht}(T) = 1 + \max_{i=1}^m \text{ht}(T_i).$$

# The number of nodes in binary trees

For a binary tree  $T$  we have the following bounds on the number  $\#nds(T)$  of nodes in  $T$ :

$$1 + ht(T) \leq \#nds(T) \leq 2^{ht(T)+1} - 1.$$

- The lower bound is attained iff  $T$  is a (single) path.
- The upper bound is attained iff  $T$  is a fully balanced tree (that is, all paths to leaves have the same length).

So one can put an exponential amount of information into a tree of a given height, and this is what one wants when using trees for data storage.

# Overview on binary search trees

*Binary search trees* are binary trees used for storing data.

- All operations for dynamic sets can be implemented.
- All these operations have running time linear in the *height* of the (current) binary search tree.

Thus it is crucial to ensure that the binary search trees constructed have low height (are close to be balanced):

In the best case all operations need logarithmic time —  
in the worst case all need linear time.

However this (how to ensure that we are close to the best case) is a more complicated topic, which is not covered in this module. (See CLRS for more information.)

# The binary-search-tree property

A **binary search tree** is a binary tree  $T$ , where we have a linear order on the universe of keys, such that

- if  $w$  is a node of  $T$  with key  $k$ ,
- then for every key  $k'$  in the left subtree of  $T$  (if it exists) we have

$$k' \leq k$$

- while for every key  $k''$  in the right subtree of  $T$  (if it exists) we have

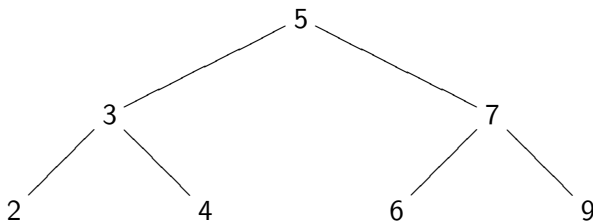
$$k \leq k''.$$

In other words:

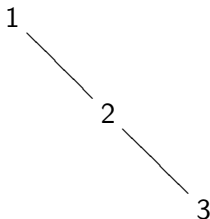
Going left the keys get smaller,  
going right the keys get larger.

# Examples

Examples for binary search trees (with integers as keys) are

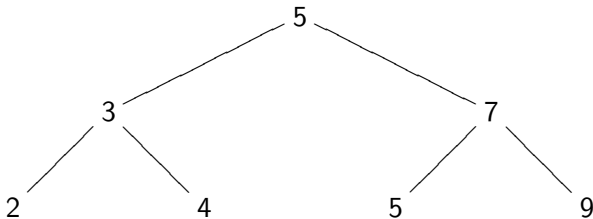


and

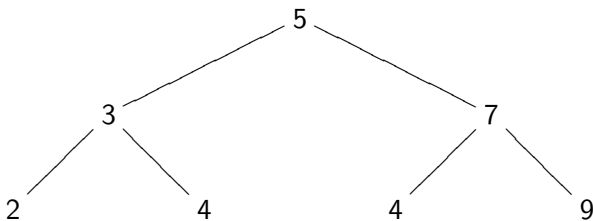


## Further examples

Still an example is



while



is not.

# Walking the tree

Our first task is for a binary search tree  $T$  to print out the keys in  $T$  in sorted order. With a recursive procedure this is very easy:

- 1 First print out the keys of the left subtree of the root.
- 2 Then print the key of the root.
- 3 Finally print out the keys of the right subtree of the root.

In other words, we just have to perform an **inorder tree traversal** (or “walk”) of  $T$ . “Inorder” here means that the root is handled in-between the two subtrees.

Since every node is traversed just once, obviously the running time of INORDER-TREE-WALK is linear in the number of nodes.

# Pseudo-code for the inorder traversal

Using a bit more precision, the parameter of INORDER-TREE-WALK shall best be

- a pointer  $x$  to some node of  $T$ , and
- the algorithm then traverses the subtree of  $T$  with root  $x$ .

Calling the procedure with the root of  $T$ , we walk the whole tree. The procedure in pseudo-code is then as follows (recall that Java-like semantics is used, and thus a pointer like  $x$  is (normally!) automatically de-referenced):

INORDER-TREE-WALK( $x$ )

```
1  if ( $x \neq \text{NIL}$ ) {  
2    INORDER-TREE-WALK( $x.\text{left}$ );  
3    print  $x.\text{key}$ ;  
4    INORDER-TREE-WALK( $x.\text{right}$ );  
5  }
```

## The linear order on the nodes

The nodes of a binary search tree are linearly ordered by the inorder-traversal, called the *infix order* — just follow the order in which the nodes are handled!

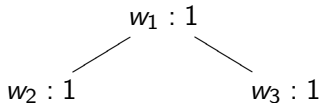
- This order is compatible with the linear order on the keys.
- So when all keys are different, then the order on the nodes is completely determined by the order on the keys.
- The infix order on the nodes does a job when it comes to nodes with the same key — here it spells out which node comes first.

Another aspect of the infix order on nodes is worth emphasising:

The infix order depends only on the tree *structure*, and no further consideration is given to the content of the nodes.

So the four order-dependent operations (min, max, successor, predecessor) are just operations on *binary trees*, ignoring keys.

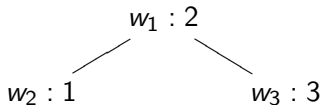
Consider for example the binary search tree



with nodes  $w_1, w_2, w_3$ , all with the same key 1. Here the infix order of the nodes is

$$w_2, w_1, w_3.$$

Note that all  $3! = 6$  possible orderings of the nodes would be compatible with the key order. This is different for



where the only order on the nodes compatible with the key order is  $w_2, w_1, w_3$ .

# Searching

Given a key  $k$  and a binary search tree  $T$  the task is to find a node with key  $k$  in  $T$  and return a pointer to it. Again, recursion is easy:

- 1 If the key  $k'$  of the root of  $T$  equals  $k$ , then return the root.
- 2 Otherwise, if  $k \leq k'$  then return the result for the left subtree of  $T$ .
- 3 Otherwise return the result for the right subtree of  $T$ .

STOP: what if  $k$  is not contained in  $T$  ?!

- The above procedure should be okay, since when coming to a leaf, then the left and right subtree will just be “empty”.
- However in the course of this action we will dereference a null-pointer, aborting the program — this must be avoided.

So we must check for a null-pointer (as before).

# Pseudo-code for searching

As before, as additional parameter we use a pointer to the root of the (sub-)tree (this enables recursion!):

TREE-SEARCH( $x, k$ )

- 1 **if** ( $x == \text{NIL}$  or  $x.\text{key} == k$ ) **return**  $x$ ;
- 2 **if** ( $k \leq x.\text{key}$ ) **return** TREE-SEARCH( $x.\text{left}, k$ );
- 3 **return** TREE-SEARCH( $x.\text{right}, k$ );

A few remarks:

- I make the pseudo-code a bit more Java/C/C++ like.
- Since as a primitive order operation only “ $\leq$ ” was assumed, we just use this relation — under reasonable assumption on key-equality and key-order this is equivalent to the use of “ $<$ ” in CLRS.

# Run time of searching

- Since in general the tree is “big”, it must be avoided that search gets “lost” in the tree.
- (This was different for the tree traversal — here we wanted to visit the whole tree.)
- Fortunately, the search just follows one path in the tree.

So the worst-case running time is  $O(\text{ht}(T))$  (linear in the height of the tree).

- If the number of nodes in  $T$  is denoted by  $n$ , then we can also say that the worst-case running time is  $O(n)$  (linear in the number of nodes).
- However this is a much weaker statement!
- For data-structure-algorithms, which run embedded in some structure, and together with other algorithms, the size of the input is typically not useful (or not even meaningful). Instead structural parameters like the height are used.

# Minimum and maximum

For the node containing the minimum resp. maximum value below a given node, we just have to follow the left-most resp. right-most path:

TREE-MINIMUM( $x$ )

- 1 **while** ( $x.\text{left} \neq \text{NIL}$ )  $x = x.\text{left}$ ;
- 2 **return**  $x$ ;

TREE-MAXIMUM( $x$ )

- 1 **while** ( $x.\text{right} \neq \text{NIL}$ )  $x = x.\text{right}$ ;
- 2 **return**  $x$ ;

Note here we have the prerequisite that  $x$  must not be a null-pointer (otherwise there is no minimum resp. maximum).

Again the worst-case running times are  $O(\text{ht}(T))$ .

# The notion of “successor”

Having a linear order  $\leq$  on a set  $M$ , an element  $x' \in M$  is a **successor** of  $x \in M$  iff

- 1  $x' > x$  (that is (for a linear order)  $x' \geq x$  and  $x' \neq x$ ),
- 2 there is no  $y \in M$  with  $y > x$  and  $y < x'$ .

If all keys of nodes in the binary search tree  $T$  are different, then we can apply this definition. However if not, then possibly amongst several nodes with the same key one has to choose.

Fortunately we have the inorder traversal of  $T$ :

For a node  $x$  of  $T$ , the **successor** of  $x$  in  $T$  is the next node in the inorder traversal of  $T$ , or NIL if  $x$  is the last element in the inorder traversal.

In other words, the successor of a node is its successor in the infix order of the nodes.

# Computing the successor

Now how to compute the successor of  $x$  in  $T$ ? (Of course, without computing the whole inorder traversal!)

- 1 If  $x$  has a right child, then by our definition (and the definition of the inorder traversal) the successor of  $x'$  is the minimum (node) of the right subtree.
- 2 But what if  $x$  has no right child? Perhaps then there is no successor??
- 3 Considering the example 8 slides ago, we see that the node with key 4 has as successor the root of the tree. Hm.

In the tutorial we will prove the following fact (please try to prove it yourself!):

## Lemma 1

*If  $x$  has no right child, then its successor is on the way up (towards the root) the first node for which we followed an edge of a left child. If there is no such node, then  $x$  has no successor.*

# Pseudo-code for successor and predecessor

## TREE-SUCCESSOR( $x$ )

```
1 if ( $x.right \neq \text{NIL}$ ) return TREE-MINIMUM( $x.right$ );  
2  $y = x.p$ ;  
3 while ( $y \neq \text{NIL}$  and  $x == y.right$ ) {  $x = y$ ;  $y = x.p$ ; }  
4 return  $y$ ;
```

## TREE-PREDECESSOR( $x$ )

```
1 if ( $x.left \neq \text{NIL}$ ) return TREE-MAXIMUM( $x.left$ );  
2  $y = x.p$ ;  
3 while ( $y \neq \text{NIL}$  and  $x == y.left$ ) {  $x = y$ ;  $y = x.p$ ; }  
4 return  $y$ ;
```

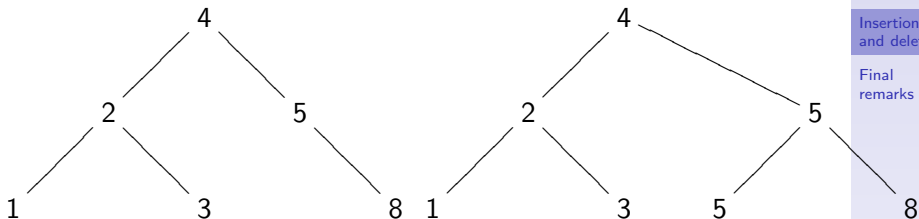
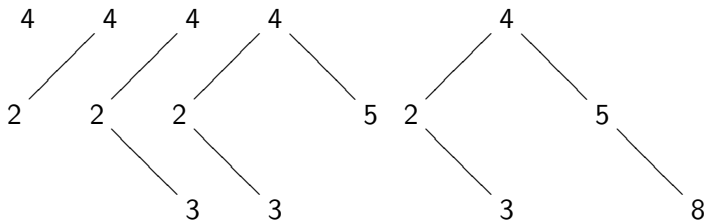
# The main idea for insertion

Consider now that a key  $k$  is given, and we want to insert it into a binary search tree  $T$ .

- There is no unique place — we can insert  $k$  into many places in  $T$ .
- But we use it “as a feature, not a bug”.
- Namely we always insert the new node as a new leaf — thus we don’t have to perform complicated tree surgery.
- That is, we follow the links to children as long as possible, that is, until we hit a node where the left or right child is missing, and where the key  $k$  can be placed as the new left resp. right child.
- The price we have to pay for this simple insertion-operation is that for “bad” input sequences we get bad (that is, very unbalanced) trees.

# An example for an insertion sequence

Inserting keys 4, 2, 3, 5, 8, 1, 5 in this order into the empty tree yields the trees



Here we have a *perfect input sequence*, creating a balanced tree.

## Pseudo-code for insertion

Instead of just the key  $k$ , we assume that a node  $z$  is given, with all links set to NIL. In this way we don't need to worry here about the creation of  $z$ .

TREE-INSERT( $T, z$ )

```
1  if ( $T.root == NIL$ ) {  $T.root = z$ ; return; }
2   $y = NIL$ ;
3   $x = T.root$ ;
4  while ( $x \neq NIL$ ) {
5     $y = x$ ;
6    if ( $x.key \leq z.key$ )  $x = x.right$ ;
7    else  $x = x.left$ ;
8  }
9   $z.p = y$ ;
10 if ( $y.key \leq z.key$ )  $y.right = z$ ;
11 else  $y.left = z$ ;
```

# Stability requirements on insertion and deletion

A general desirable requirement on insertion and deletion is, that these operations should only alter the infix order as directly related to the action:

- That is, insertion only adds a further node, while deletion keeps all other nodes (except of the one being deleted).
- Since insertion and deletion must produce a valid binary search tree, preservation of the infix order means that the order of nodes with equal keys is preserved.

Now one should strengthen the requirement on minimal alteration of the infix order by demanding that **all existing pointer (iterators)** into the tree **shall stay valid** (with the exception of the node being deleted).

- 1 “Validity” must include that the observable content of nodes, key and satellite data, is unaltered.
- 2 However links to parents and children may be altered.

# Analysing deletion

Our insertion operation obviously fulfils the stability requirements. We will now use these requirements to guide our development of the deletion algorithm for deleting node  $z$  in  $T$ .

A general problem-solving strategy is to start always with the easy cases:

- 1 If  $z$  has no children, then  $z$  can be just deleted, and for its parent the respective child-link is set to NIL.
- 2 If  $z$  has only one child, then again  $z$  can be just deleted, while the child-link of  $z$ 's parent is re-linked to that single child of  $z$ .

So the remaining case is that  $z$  has two children, call them  $a$  and  $b$ :  $z$ 's parent has only one (child-)link for  $z$ , not two as would be required to hold  $a$  and  $b$ .

## The main idea for deletion

Could we just replace  $z$  with another node  $z'$  from  $T$  ?!

- If nothing else shall be changed, then  $z'$  must still be greater (or equal) than all nodes in the left subtree of  $z$ , and smaller (or equal) than all nodes in the right subtree of  $z$ .
- To keep the infix order, only the successor of  $z$  (in the infix order) is eligible, and actually fulfils the order requirements!

So the idea is to take the successor  $z'$  of  $z$ , which by our analysis of the successor operation is element of the right subtree of  $z$ , and put it into the place where  $z$  is. But stop — don't we have the same problem with  $z'$ , which can not be taken out so easily if  $z'$  has two children ?

Fortunately,  $z'$  is the leftmost node in the right subtree, and thus *has no left child*.

So we can relink  $z'$ 's parent with the only child of  $z'$ , and put node  $z'$  in the place of  $z$ .

# Final details of deletion

The implementation of deletion now is straightforward, one only has to take care of the details; please see CLRS.

- Keep in mind that we keep old nodes, and thus  $z'$  is *relinked* with the children of  $z$  and its parent.
- The alternative, as actually to be found in the earlier editions of CLRS, to put the content of  $z'$  into node  $z$ , violates the stability requirements, since node  $z'$  would in fact be deleted, not node  $z$ .
- The code is slightly lengthier due to the left/right cases.
- The organisation in the book requires to consider the special case where  $z'$  is  $b$  (the right child of  $z$ ), to get the relinking right (in their organisation).

## Bad orders are possible

The framework for binary search trees now is a reasonable starting point.

- However, bad orders exist!
- For example strictly increasing or strictly decreasing sequences of key insertions.
- Then the binary trees degenerates to just a path.
- So we get time complexities  $O(n)$  for all operations, not  $O(\log n)$  as would be the case if the tree would be reasonably balanced ( $n$  is the number of nodes of the tree).

There are two general responses to such a situation:

- 1 Hope for randomisation.
- 2 Or improve the worst-case behaviour, by using more intelligent algorithms.

See CLRS for more on the second strategy.

## Logarithmic height “on average”

As demonstrated in the lab session, if input sequences (insertion sequences) are “shuffled”, then the height grows (empirically) only logarithmic.

- This is proven in Subsection 12.4 of CLRS, where it is shown that the average height of binary search trees, over all possible insertion sequences of length  $n$  (all equally likely) is  $O(\log n)$ .
- Of course it is in fact  $\Omega(\log n)$ .
- In general, one needs to be careful in such situations, where the average behaviour is good, which however might not be the *typical behaviour* — this is called the “lottery phenomenon”, where big outliers dominate the average.
- Fortunately, here this is not the case, since the average is “low” (not “high”), and thus the average behaviour is also the typical behaviour.

# Improving multiple insertions

One aspect of the insertion algorithm can be easily improved, namely when inserting a sequence of equal keys:

- 1 Yet we produce a degenerated tree (the worst-case), just growing to the left.
- 2 This is due to the fact that in the existing inorder sequence of equal-key-nodes, the new node inserted is inserted in the first possible position.
- 3 One could also insert it into last possible position, always going right — however this wouldn't improve anything.
- 4 The right thing to do is to do both, in the right mixture!

See Problem 12.1 in CLRS for possible strategies (easiest to implement is to randomise the left/right choices).