Föreläsning 16

Splay-trees, Heaps, Skip-lists

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
18 November 2024

IDA, Linköpings universitet

16.1

Content

Contents

16.2

1 Splay-trees

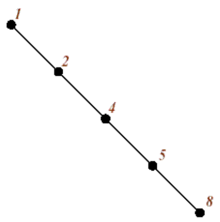
Binary search trees are not unique

Recall that binary search trees:

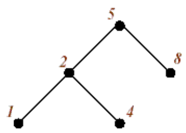
- Allow for simple insertion and deletion, but ...
- ”balance” depends on insertion and deletion orders.

Combine with the heuristic: ”keep last used first”?

- Elements that currently most often used should be close to the root!



insert: 1,2,4,5,8



insert: 5,2,1,4,8

16.3

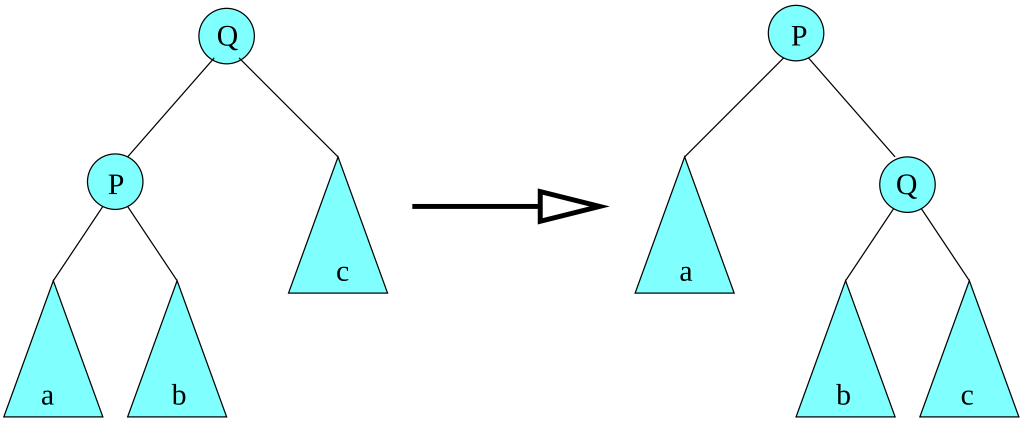
Operation splay(k)

- Perform a normal search for k, and remember the nodes we pass...
- Let P be the last node we visit
  - If k is in the tree T, then it is in P,
  - otherwise, P is parent to an empty node in the tree
- Get back to the root and perform a rotation at each node to move P up in the tree ... (3 cases)

16.4

Operation splay(k)

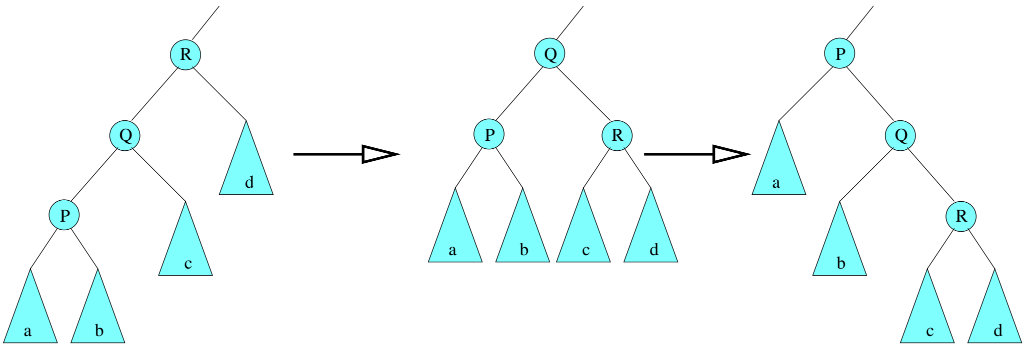
- zig: parent(P) is root: rotate wrt. P



16.5

Operation splay(k)

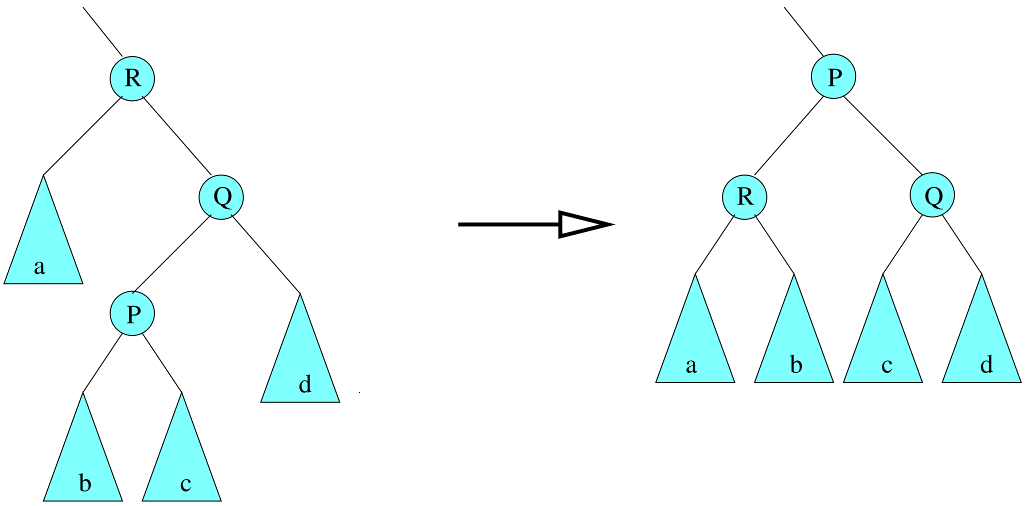
- zig-zig: P and parent(P) are both left children (or both right children): perform two rotations to move P upwards



16.6

Operation splay(k)

- zig-zag: One of P and parent(P) is a left child and the other is a right child: perform two different rotations



Observe that rotations can increase the height of the tree!

16.7

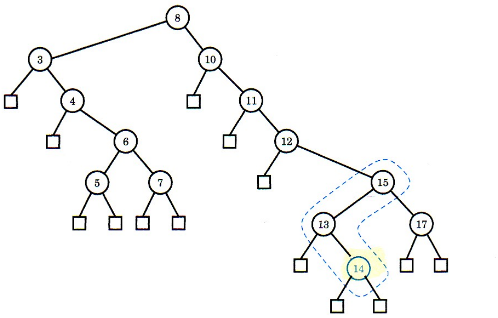
find and insert

function FIND(k,T)
  SPLAY(k,T)
  if KEY(ROOT$(T)) = k$ **then return** (k,v)
  else return null

function INSERT(k,v,T)
  insert (k,v) as in a binary search tree
  SPLAY(k,T)

16.8

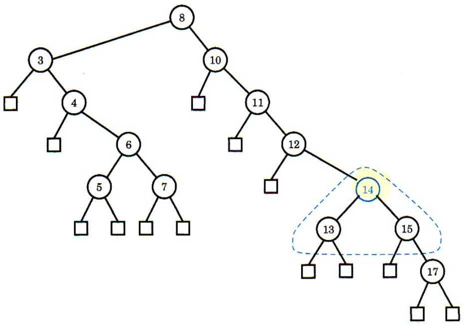
Example: insertion of 14



(a)

16.9

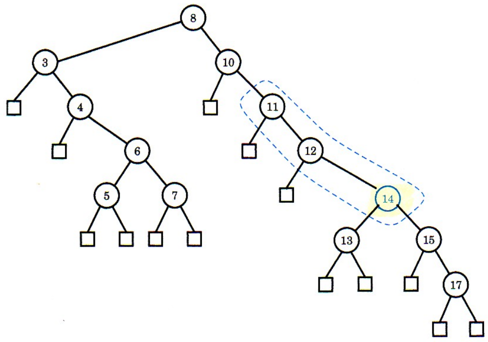
Example: insertion 14



(b)

16.10

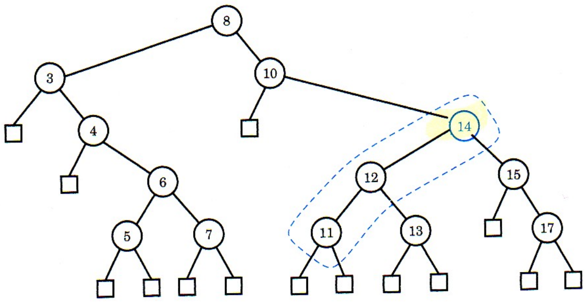
Example: insertion of 14



(c)

16.11

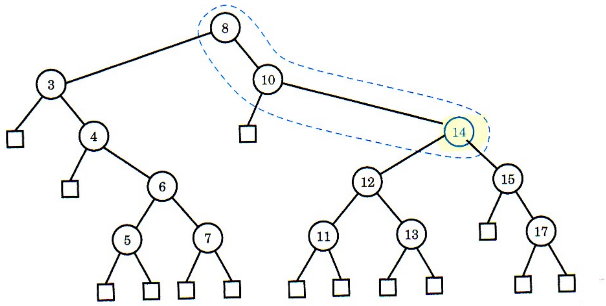
Example: insertion of 14



(d)

16.12

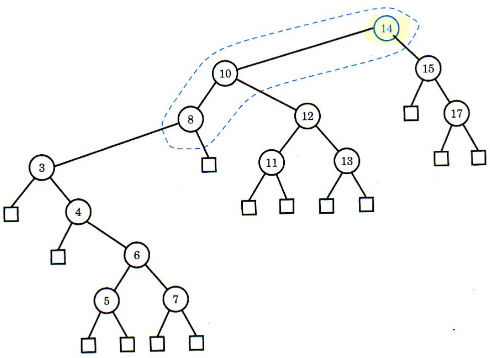
Example: insertion of 14



(e)

16.13

Example: insertion of 14



(f)

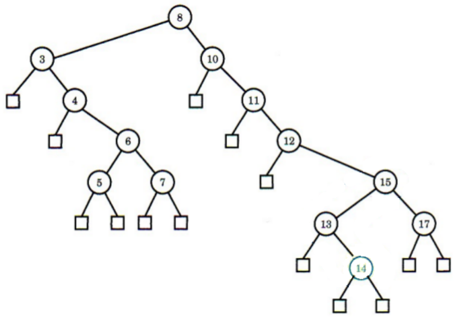
16.14

delete

function DELETE(k, T)
  SPLAY(k, T)
  if KEY$($ROOT$(T)) = k$ **then**
    remove ROOT(T): gives T_{left} and T_{right}
    do SPLAY on max value in T_{left}, gives T'_{left}
    bind T_{right} to ROOT(T'_{left})

You can also use successor in inorder traversal. 16.15

Example: remove 8



16.16

Performance

- Each operation might need to be executed on unbalanced trees
  - no guaranty to achieve $O(\log n)$ in worst case

- Amortized time is logarithmic
  - each sequence of m operations, executed on an initially empty tree, take in total $O(m \log m)$ time complexity
  - therefore, the *amortized* cost for an operation is $O(\log n)$ even if individual operations can perform much worst

16.17

2 Priority queues

Priority queues

Naturally encountered:

- waiting lists (among tasks, events in a simulation)
- If a resource is free, choose an element from the waiting list
- Choice based on a partial/linear order:
  - task with highest priority is chosen
  - each event is to occur at some time, events are to be processed in time order

16.18

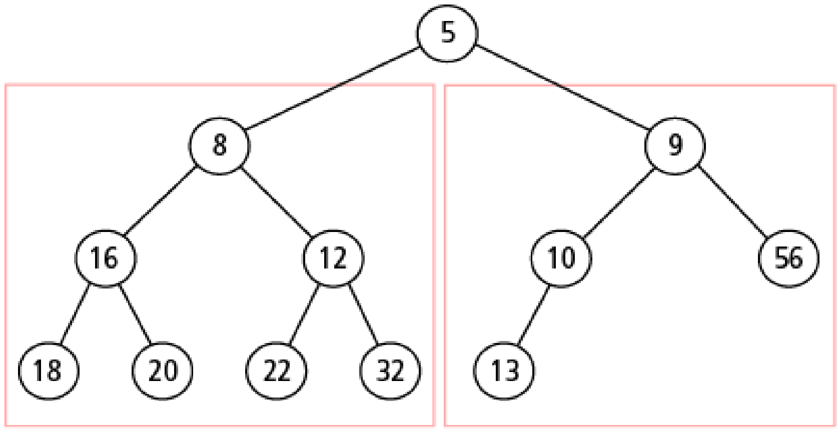
ADT priority queue

- Linearly ordered set of keys K
- We store pair (k, v) (as in a dictionary ADT), *multiple pairs* with *same key* are allowed
- A typical operation is to fetch a pair with a minimal key
- Operations on a priority queue P:
  - makeEmptyPQ()
  - isEmpty()
  - size()
  - min(): find pair (k, v) with minimal k in P; return (k, v)
  - insert(k, v): insert (k, v) in P
  - removeMin(): remove and return a pair (k, v) in P with a minimal k; **error** if P is empty

16.19

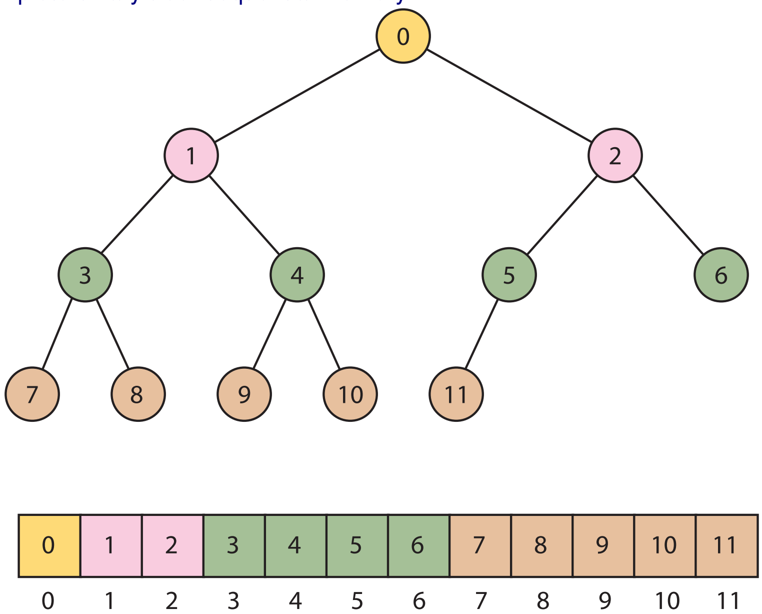
Implementation of priority queues

- One could use sorted linked lists or BSTs
- Another idea: use a complete binary tree where the root, in each (sub)tree, contains a minimal element in the (sub)tree



This is a partially sorted tree, also called a heap! 16.20

Complete binary tree: sequential memory

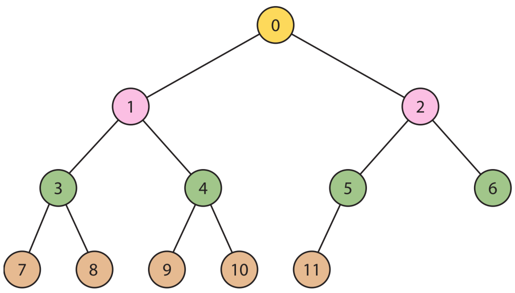


16.21

Sequential memory

Use a table table<key,info>[0..n-1]

- leftChild$(i) = 2i+1$ (returns **null** if $2i+1 \geq n$)
- rightChild$(i) = 2i+2$ (returns **null** if $2i+2 \geq n$)
- isLeaf$(i) = (i < n)$ and $(2i+1 > n)$
- leftSibling$(i) = i-1$ (returns **null** if $i = 0$ or odd(i))
- rightSibling$(i) = i+1$ (returns **null** if $i = n-1$ or even(i))
- parent$(i) = \lfloor (i-1)/2 \rfloor$ (returns **null** if $i = 0$)
- isRoot$(i) = (i = 0)$



16.22

2.1 Heaps

Updating a heap structure

- The last leaf is the last node when traversing level by level

- removeMin(PQ) *// remove the root*
  - Replace root with last leaf
  - Restore the partial ordering by pushing the node downwards with ”down-heap bubbling”

- insert(PQ, k, v)
  - insert node (k, v) after the last leaf
  - Restore the partial order with ”up-heap bubbling” 16.23

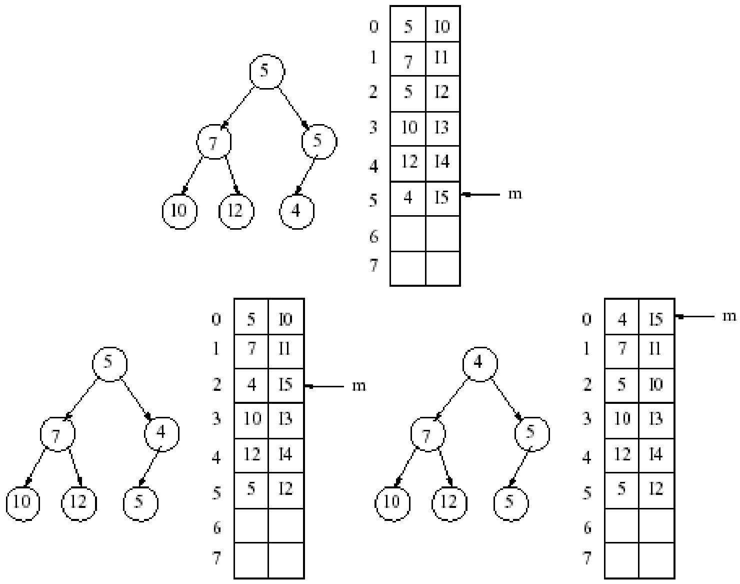
Properties

- size(), isEmpty(), min(): $O(1)$
- insert(), removeMin(): $O(\log n)$

Recall array representation of a complete binary tree ...

- Compact representation
- ”Bubble-up” and ”bubble-down” have efficient implementations 16.24

Example: ”bubble-up” after insert(4,15)



16.25

Recall ArrayList from lecture 7

- Write a class that implements an array of integers
  - We call it `ArrayList`
  - Behavior:

```
add(value)         insert(index, value)
get(index)         set(index, value)
size()             isEmpty()
remove(index)
indexOf(value)     contains(value)
toString()
...
```

- The size of the list will be the number of elements inserted so far
  - The actual length of the array (capacity) can be larger. Start with a size of 10 by default. 16.26

Destructor

- ```
  // ClassName.h       // ClassName.cpp
  ~ClassName();        ClassName::~ClassName() { ...
  ```
  - Called when the object is destroyed by the program (when the object goes out of scope or delete is used)
  - Can be useful to:
    - free temporary resources
    - free dynamically allocated memory used by the members
- Does `ArrayList` need a destructor? What should it do?
  - Yes; to free the memory associated with storing elements

16.27

Increase capacity

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	4	8	1	6
size	10	capacity	10							

- What if the users wants to add more than ten elements?

  ```
  list.add(75) //add a 11th element
  ```

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	3	8	9	7	5	12	4	8	1	6	**75**	0	0	0	0	0	0	0	0	0
size	**11**	capacity	**20**																	

- Answer: double the size of the field
  - Do not forget to release the memory used by the old array!
  - ```
    int* a = new int[10];
    int* b = new int[20];
    std::copy(a, a+10, b); // Do not use memcpy(b, a, 10 * sizeof(int))!
    delete[] a;
    a = b;
    std::copy(first, after, output);
    ```

16.28

Amortised analysis

We want a new type of array that automatically increase available size when full (when the number of ellementsis n is same as the capacity N). Suppose the array always insert new element in the first free position:

- Allocate a new array B with capacity $2N$
- Copy $A[i]$ to $B[i]$, for $i = 0, \dots, N-1$
- Lets $A = B$, we let B take over the role A had.

In term of effectiveness, expanding the array is slow. But the algorithmic complexity is:

- $O(1)$ most of the time
- $O(n)$ for copying n element and $O(1)$ for inserting after reallocation.

16.29

3 Skip-lists

Skip-lists

- A hierarchical linked list...
- A randomized alternative to implementing a dictionary ADT
- Insertion uses randomization ("coin tossing")
- Good expected performance
- Worst behavior occurs extremely rarely (for more than 250 data elements, the risk that the search time is more than 3 times the expected time is less than 10^{-6})

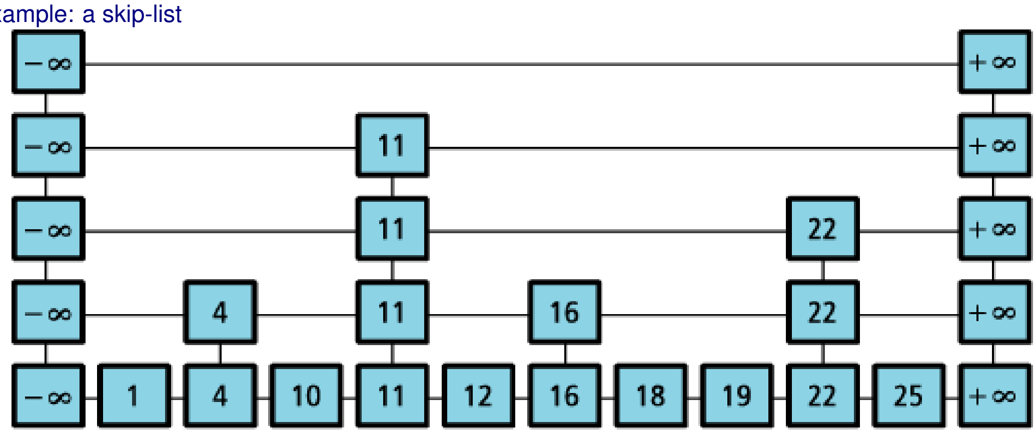
16.30

The skip-list data-structure

- Levels $L_1, \ldots, L_h$ of nodes (keys, values)
- Same nodes on several levels (tower)
- Special keys: $-\infty$ and $+\infty$ ... smaller/larger than all real keys...
- Several *levels* of doubly linked lists, the higher the sparser
  - Level 1: all nodes are part of a doubly linked list from $-\infty$ to $+\infty$, ordered according to '<'-relation
  - In average, half the nodes from L_i are also part of L_{i+1}
  - Special keys $-\infty$ and $+\infty$ are part of all levels
  - Only $-\infty$ and $+\infty$ are part of L_h

16.31

Example: a skip-list



16.32

Search

Seach key k:

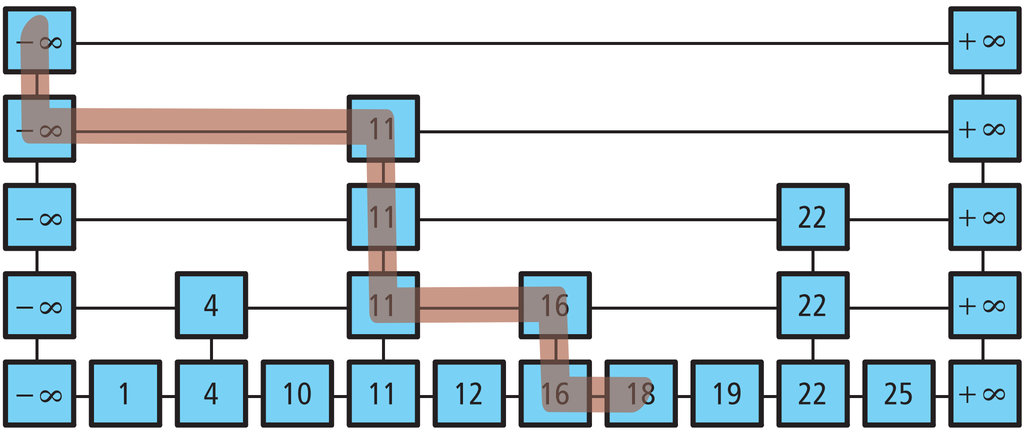
- Follow the list at the highest level...
  - Stop just before passing some $k_i > k$
  - If found k, return the result, otherwise ...
- We stopped at some level:
  - Did we find the key?
  - No, change the lower level (using the "last tower") and continue searching
  - Return: largest key $k_i \leq k$ (which could be $+\infty$)

16.33

Searching

Searching for key k:

- Similarities with binary search, but for lists
- Example: find(18)



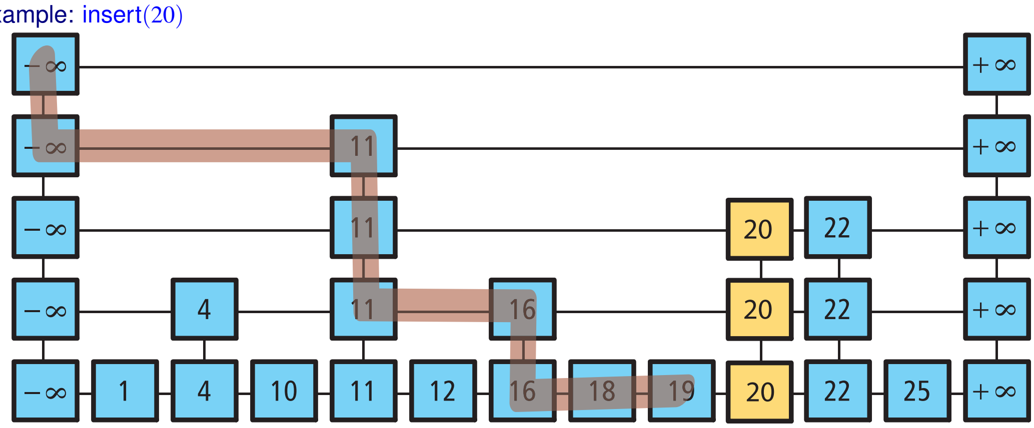
16.34

Insert

function INSERT(x)
    $P \leftarrow$ FIND(x)
    if $P.value < x$ **then**
        insert a new node after P
        ”toss a coin” to decide how high the ”tower” should be:
        while ”tossing a coined”=yes **do**
            increase tower with one level
            (might increase the height of the skip-list)

16.35

Example: insert(20)



16.36

Deletion . . . and properties

- Similar to search:
  - Search
  - if found, remove and repair links between the towers
- Worst case for find, insert and remove in a skip-list with n elements is $O(n+h)$
- But expected execution time (assuming the keys are uniformly distributed) is $O(\log n)$ if the search starts at height $\lfloor \log n \rfloor$

16.37