Lecture 18

Splay-trees, hashing, skip-lists

TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
11 november 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

Content

Innehåll

1 Splay-trees

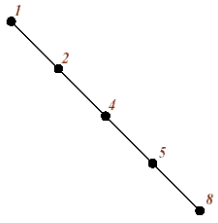
Binary search trees are not unique

Remember the binary search trees:

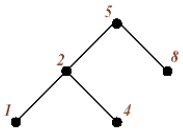
- Easy to insert and remove elements, but...
- "balance" is determined by the order of insertions and deletions.

Combine with heuristics "hold recently used elements first" for lists?

- Often, use elements close to the root!



insert: 1,2,4,5,8



insert: 5,2,1,4,8

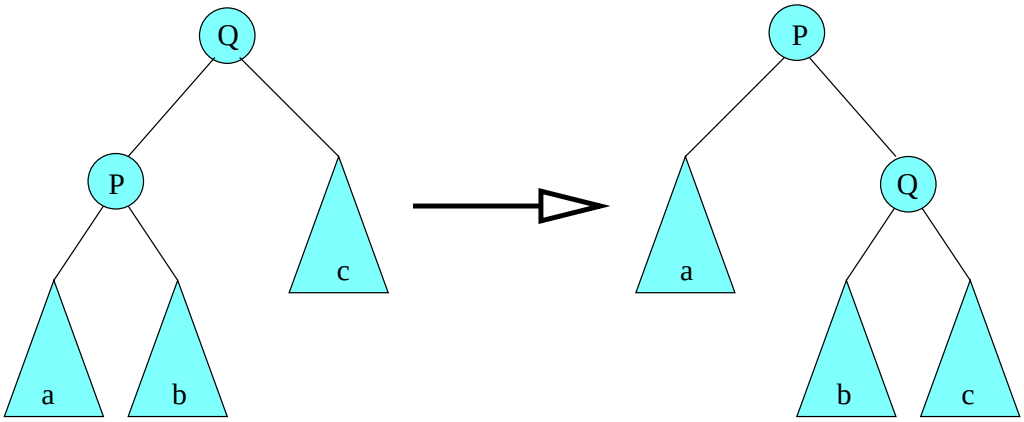
The operation splay(k)

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again

- Perform a normal search for k, remember the nodes we pass...
- Mark the last node we examine with P
  - If k exists in T, it must be in node P,
  - otherwise, P is a parent of an empty tree.
- Return to the root and do a rotation at each node to move P upwards in the tree... (3 cases)

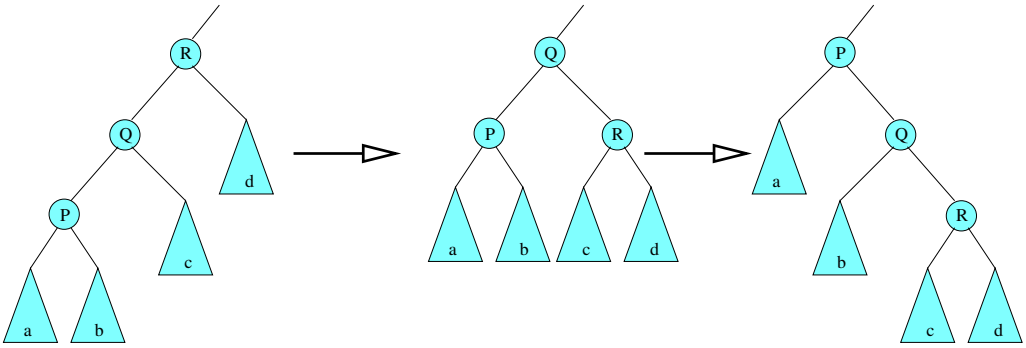
The operation splay(k)

- zig: parent(P) is the root: rotate around P



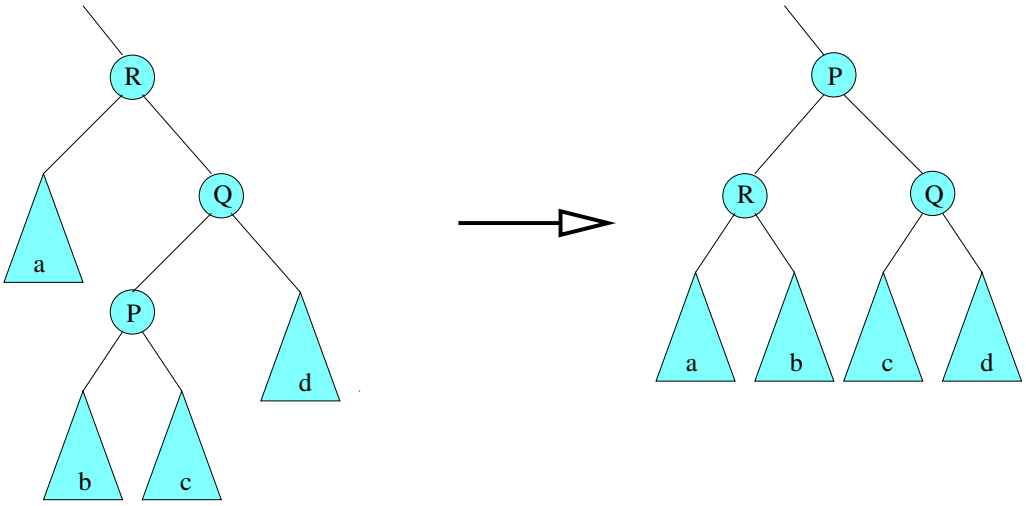
The operation splay(k)

- zig-zig: P and parent(P) are both left children (or both right children): perform 2 rotations to move up P



The operation splay(k)

- zig-zag: One of P and parent(P) is a left child and the other is a right child or vice versa: perform 2 rotations in different directions.



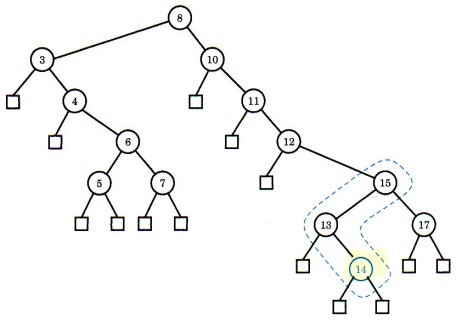
Note that these rotations can increase the height of the tree!

find and insert

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

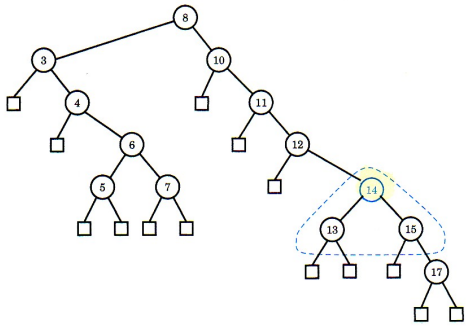
function INSERT(k, v, T)
  insert (k, v) in a binary search tree
  SPLAY(k, T)

Example: insertion of 14



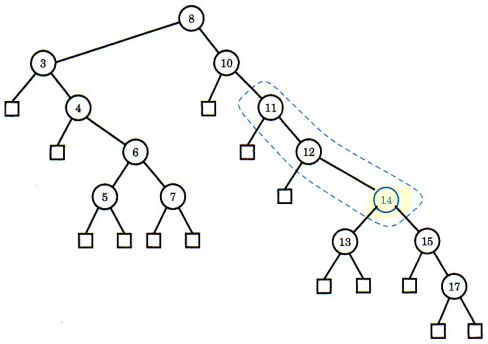
(a)

Example: insertion of 14



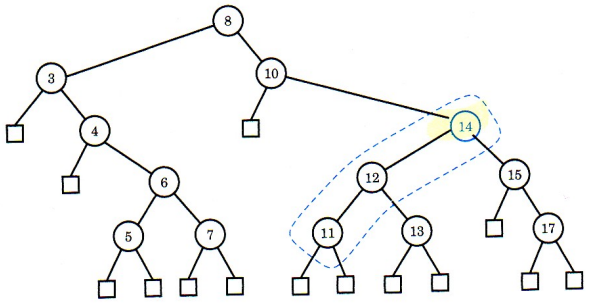
(b)

Example: insertion of 14



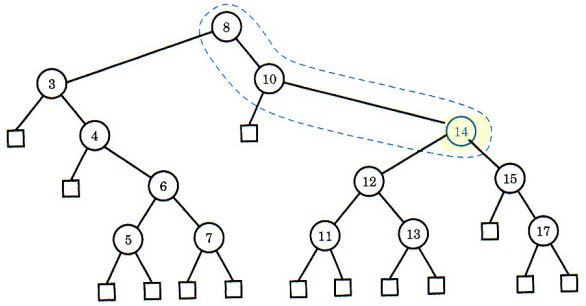
(c)

Example: insertion of 14



(d)

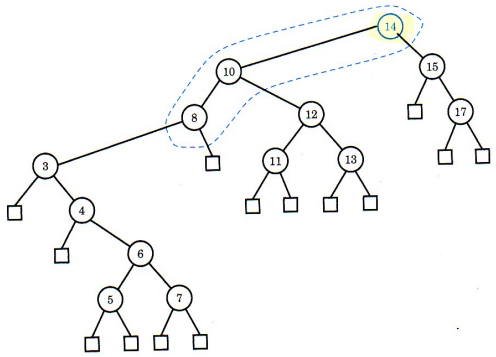
Example: insertion of 14



(e)

18.13

Example: insertion of 14



(f)

18.14

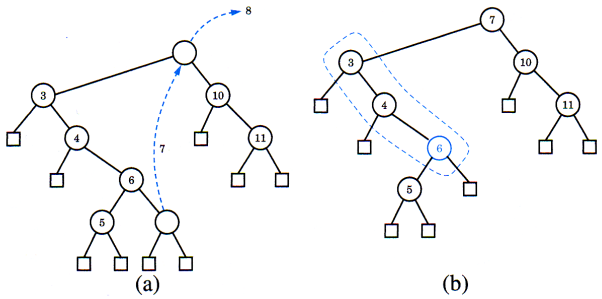
delete

function DELETE(k, T)
  if k is located in a leaf **then**
    do SPLAY to the parent of the leaf
  else if k is located in an internal node **then**
    replace the node with its predecessor in the order
    do SPLAY to the parent of the predecessor

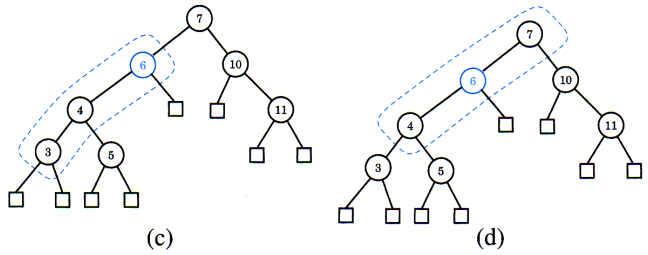
It is of course possible to use the successor in the order also.

18.15

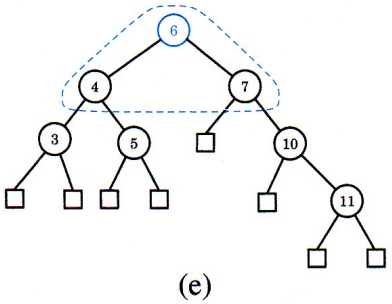
Example: removing 8



Example: removing 8



Example: removing 8



(e)

18.18

Performance

- Each operation may need to be carried out in a totally unbalanced tree
  - thus, no guarantee for $O(\log n)$ time in the worst case

- The amortized time is logarithmic
  - Each sequence of m operations performed on an initially empty tree lasts for $O(m \log m)$ time.
  - thus, the *amortized* cost/time for an operation is $O(\log n)$ although individual operations can behave much worse

18.19

2 Hash tables

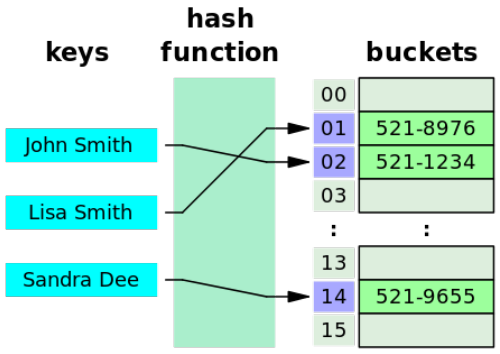
Can we find something better?

yes, with the help of *hash tables*

- Idea: given a table $T[0,\ldots,max]$ storing elements ... ...*find a suitable table index* for each element
- Find a function h such that $h(key) \in [0,\ldots,max]$ and (ideally) $k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$
- Store each pair key-value (k,v) in $T[h(k)]$

18.20

Hash tables



18.21

Hash tables

- In practice, hash functions are not given unique values (they are not *injective*)
- We need collision handling

... and

- We need to find a good hash function

18.22

2.1 Collision Handling

Collision Handling

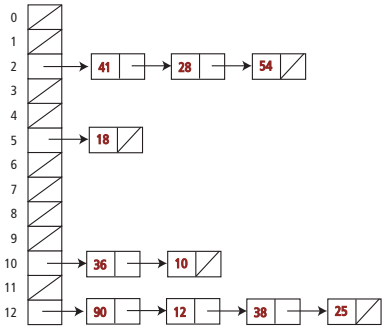
Collision occurs when at least 2 keys are hashed to the same slot. 2 principles to handle collisions:

- *Linking*: keep colliding data in linked lists
  - *Separate linking*: having the linked lists off the table
  - *Collective linking*: store all data *in* the table
- *Open adressing*: store all data *in* the table *and* let some algorithm to decide which index to use in a collision

[Eng: Separate Chaining, Coalesced Chaining, Open Addressing] 18.23

Example: hashing with separate linking

- Hash table of size 13
- Hash function h with $h(k) = k \bmod 13$
- Sore 10 integer keys: 54, 10, 18, 25, 28, 41, 38, 36, 12, 90



18.24

Separate linking: find

Given: key k, hash table T, hash function h

- calculate $h(k)$
- look for k in the pointing out list $T[h(k)]$

Notation: probe= an access to the linked list

- 1 probe to access the list header (if non-empty)
- 1+1 probing to access the contents of the first list element
- 1+2 probing to access the contents of the second list element
- ...

A probe (to follow a pointer) takes a constant time. 18.25

Separate linking: failed lookups

- n data elements
- m entries in the table

Worst case:

- all data elements have the hash value: $P = 1 + n$

Average case:

- hash values uniformly distributed over m:
- average length α of the list: $\alpha = n/m$
- $P = 1 + \alpha$

18.26

Separate linking: successful lookups

Average case:

- accessing $T[h(k)]$ (the first element of a list L): 1
- traversing $L \Rightarrow k$ found after: $|L|/2$
- expected $|L|$ corresponding to α, thus: expected $P = \alpha/2 + 1$

18.27

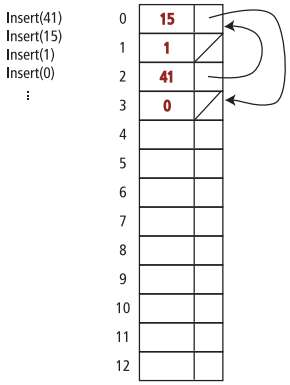
Collective linking: keep an element in the table

- Place the data element in the table
- Extend them with pointers
- Solve collisions by using the first free places

Chains may contain keys with different hash values. . . . . but all keys with the same hash value show up in the same chain

+ better memory utilization - The table can get full -Longer collision chains

Collective linking



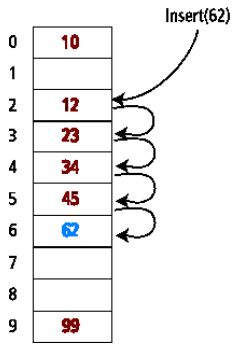
Open addressing

- Store all elements inside the table
- Use a fixed algorithm to find a vacant place

Sequential/linear probing

- desirable hash index $j = h(k)$ (mod 10 here)
- if conflict arises go to the *next* free place
- If the last element of the table reached, go to the first element. . .

- Positions near to each other fill up quickly (*primary clustering*)
- How to make remove(k)?



Open addressing — remove()

The element to be removed can be part of a collision chain – can we determine it?

If it is part of a chain, we can not only remove the element!

- Since all keys are stored, hash all of the the data left?
- Look through the elements after hashing or drawing together, when appropriate, and stop at the first available position. . .
- Ignore – insert a mark deleted"if the next place is not empty. . .

Double hashing – or what to do in collision?

- Second hash function h_2 calculates *increment* in case of conflicts
- When Increment value is out of the table, we use modulo $m = tableSize$

Linear probing is a double hashing with $h_2(k) = 1$ Requirements for h_2:

- $h_2(k) \neq 0$ for all k
- $h_2(k)$ has no common divisor with m for any $k \Rightarrow$ *all* table positions can be reached

A common choice $h_2(k) = q - (k \bmod q)$ for $q < m$, m prime number (i.e. choose a prime number less than the table size!)

2.2 Choosing a hash function

What is a good hash function?

Assuming that k is a natural number.

Hashing provides a uniform distribution of hash values, *but* this depends on the *distribution of keys* in the data to be hashed.

Example: Hashing of the surnames in a group of students

- hash function: ASCII-value of the last letter *bad choice:* the majority of names end with 'n'.

String hashing in Java

`hashCode()` for `String` i Java 1.1

- For long strings: examine just 8-9 evenly placed characters.

```
public int hashCode()
{
   int hash = 0;
   int skip = Math.max(1, length() / 8);
   for (int i = 0; i < length(); i += skip)
      hash = s[i] + (37 * hash);
   return hash;
}
```

- Advantage: save time
- Disadvantage: great potential for bad collision patterns

Suggestion for hash functions

- Memory address
  - Interpreting the memory address where the object to be hashed is stored in as an integer
  - Works well in general, but not good for example for numeric keys or string keys.
- Converting to integer
  - Interpreting the bits of the key as an integer
  - Suitable for keys of shorter length than the number of bits in integers
- Component sum
  - Divide the bits of the key into components of fixed length (eg 16 or 32 bits) and sum the components. (Ignore the overflow.)
  - Suitable for numeric keys of constant length greater than or equal to the number of bits in the integer type.

Suggestion for hash functions

- polynomial accumulation
  - Divide the bits of the key in a sequence of components of fixed length (e.g. 8, 16 or 32 bits)

$$a_0 a_1 .. a_{n-1}$$

  - Evaluate the polynom

$$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$

    with a fixed value z. (Ignore overflow.)
  - Very suitable for hashing the strings. (e.g. $z = 33$ gives at most 6 collisions on an amount of 50000 English words.)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horners rule:
  - The following polynomial calculated successively. Each polynomial i the sequence can be calculated in $O(1)$ time from the previous polynomial in the sequence.

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \ (i = 1, 2, \ldots, n-1)$$

- We have $p(z) = p_{n-1}(z)$

18.36

String hashing in Java

`hashCode()` for `String` in Java nowadays

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i:te tecknet i s

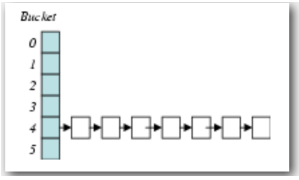
char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

18.37

Algorithmic complexity attacks

Player assumption of a uniform distribution of keys to insert any role in practice?

- Obvious situations: air traffic, nuclear power plants, pacemaker
- Surprisingly situations: denial-of-service-attacker



Elak motståndare lär sig din hashfunktion (t.ex. genom att läsa Javas API) och orsakar en lång kö i enstaka cell vilket påverkar prestanda drastiskt.

Real attack opportunities [Crosby-Wallach 2003]

- Bro server: send carefully selected packages to (DOS-) attack the server with less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully selected strings in the associative array.
- Linux 2.4.20-core: save files with thoroughly selected names.

18.38

Algorithmic complexity attacks in Java

- Goal: Find a family of strings with the same hash value.
- Solution: Javas string-API uses 31-base code for string hashing

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strängar av längd 2N som hashar till samma värde!

18.39

Hashing by integer division

Let m be the table size

$$h(k) = k \bmod m$$

Avoid

- $m = 2^d$: hashing gives the last d bits in k
- $m = 10^d$: hashing gives the last d numbers

It is usually suggested to use prime numbers for m Examine samples from real data to experiment with hashing parameters.

See `http://burtleburtle.net/bob/hash/doobs.html` for other opinions.

18.40

3 Skip-lists

Skip-lists

- A hierarchical linked list...
- A randomized alternative to implement ADT Dictionary
- Insertion uses randomization (”coin toss”)
- Good performance in the expected case
- Worst case performance of skip-lists occurs very rarely (>250 data element, the risk that the search time is more than 3 times than expected is under 10^{-6})

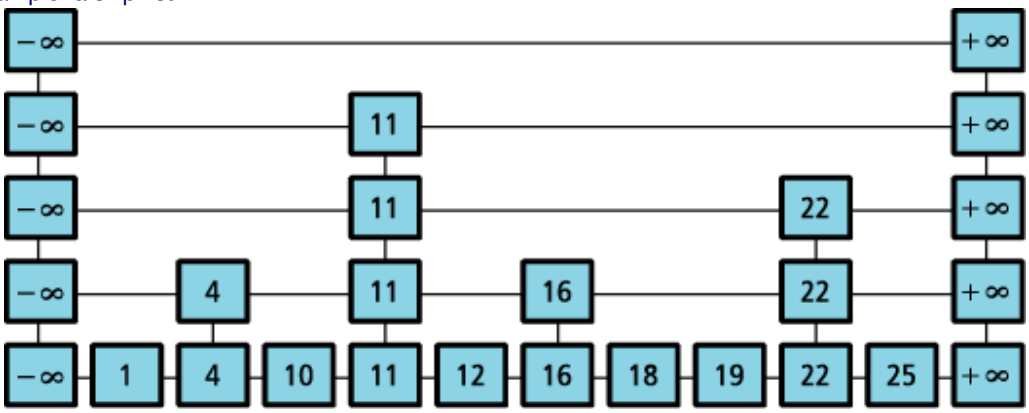
18.41

Skip-list data structure

- Levels $L_1, \ldots, L_h$ of nodes (keys, values)
- Same nodes exist in different -several- levels (tower)
- Special keys: $-\infty$ and $+\infty$ ...less/greater than each real key...
- Several *levels* of doubly linked lists, sparser higher
  - Level 1: all nodes in a doubly-linked list between $-\infty$ and $+\infty$ arranged according to ’<’-relation
  - In average, one half of the nodes exists in L_i and also in L_{i+1}
  - Special keys $-\infty$ and $+\infty$ exist in all levels
  - Only $-\infty$ and $+\infty$ exist in level L_h

18.42

Example: a skip-list



18.43

Search

Search for key k:

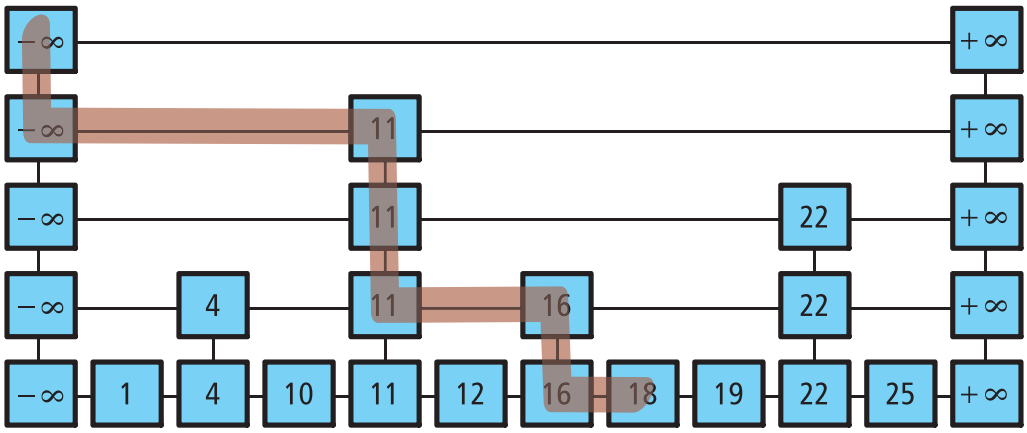
- Follow the list of highest level...
  - Stop before reaching any $k_i > k$ (we risk to miss what we're looking for)
  - If we found the right, return it, otherwise...
- We have been searching in one level:
  - Have we found the key?
  - No, change to next lower level (via "the last tower") and keep looking
  - Returns: the largest key $k_i \leq k$ (which can be $+\infty$)

18.44

Search

Search for key k:

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



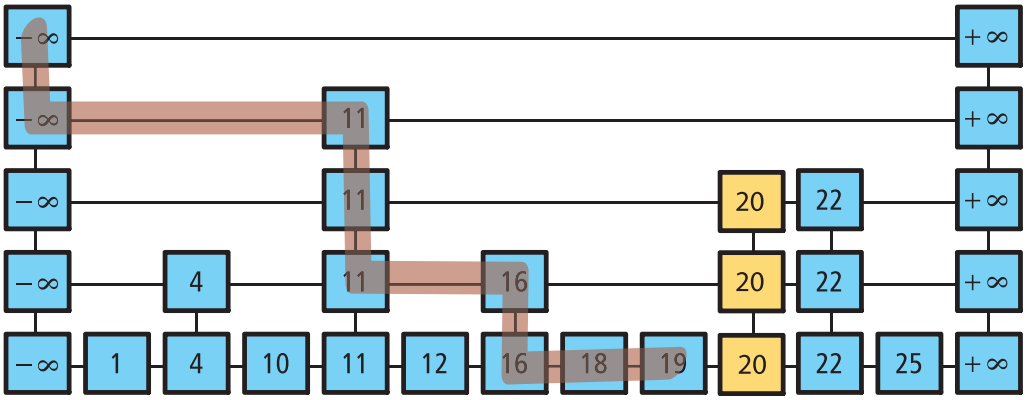
18.45

Insertion

function INSERT(x)
  $P \leftarrow$ FIND(x)
  if $P.value < x$ **then**
    insert a new list node after P
    "toss a coin" to determine how high the "tower" is:
    while "toss"=yes **do**
      increase the tower height with one step
      (increase possibly the height of the skip-list)

18.46

Example: insert(20)



18.47

Removing... and characteristics

- Similar insert:
  - Search
  - if found, remove it and fix the links between the towers
- The worst case time for find, insert and remove in a skip-list with n inserted element is $O(n+h)$
- However, the expected execution time (assuming that the keys are uniformly distributed) is $O(\log n)$ if the search starts at height $\lfloor \log n \rfloor$

18.48