Föreläsning 17

Hashing, Tree-based applications

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
14 november 2023

IDA, Linköpings universitet

17.1

Content

Contents

17.2

1 Hash tables

Can we find something better than BST for sets?

Yes, with *hash tables*

- Idea: given a table $T[0,\ldots,max]$ to store the elements … …*find a 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 key-value pair (k,v) in $T[h(k)]$

17.3

Hash table

- In practice the hash functions do not give unique values (they are not injective)
- We need to manage collisions

…and

- We need to find a good hash function

17.4

1.1 Collision management

Collision management

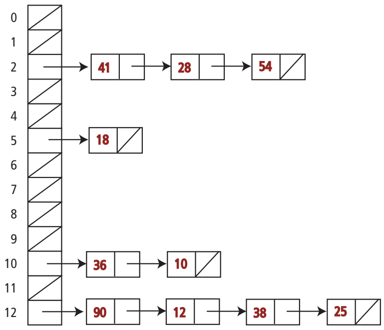
Two approaches for managing collisions:

- *Open hashing* or *separate chaining*. Maintain colliding data outside the table, e.g., using linked lists.
- *Closed hashing* or *open addressing*. Store all data in the table and let some algorithm decide which index to use in case of a collision.

17.5

Example: hashing with separate chaining

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



17.6

Separate chaining: find

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

- compute $h(k)$
- look-up k in the list $T[h(k)]$

Notation: probing= an access in the linked list

- 1 probing to access the head of the list(if non-empty)
- 1+1 probing to access the content of the first element
- 1+2 probing to access the content of the second element
- ...

A probing (to follow a pointer) takes constant time. How many probings P are needed to get a data in the hash-table?

17.7

Separate chaining: unsuccessful look-up

- n data elements
- m locations in the table

Worst case:

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

Average case:

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

17.8

Separate chaining: successful look-up

Average case:

- access to $T[h(k)]$ (beginning of the list L): 1
- traverse $L \Rightarrow k$ if found after: $|L|/2$
- expected $|L|$ corresponds to α, so: expected $P = \alpha/2 + 1$

17.9

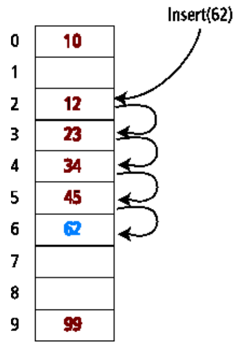
Open addressing

- Store all elements inside the table
- Adopt a fixed algorithm to find a free place

Linear probing

- targeted hash index $j = h(k)$
- if conflict, move to next available position
- if reach end of the table, go to the start...

- Positions next to each other become full (*primary clustering*)
- How to remove(k)?



17.10

Open addressing — remove()

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

If it is part of a collision chain, we can not just remove it!

- Rehash all keys?
- Check following elements in the list and rehash until hit first free position ... ?
- Ignore – place a marker "deleted" if next place is not empty...

17.11

What to do in case of collision?

- Linear probing by steps hash function $h(K) + i \times c$ computes an *increment* in case of a collision
- Quadratic probing hash function $h(K) + c_1 \times i^2 + c_2 \times i + c_3$
- A second hash function $h_2(K, i)$ computes an *increment* based on the step and the key

Linear probing is double hashing with $h_2(k, i) = i \times c$ Requirements on h_2:

- $h_2(k, i) \neq 0$ for all k
- $h_2(k, i)$ should go through all positions in the table by iterating through i. E.g., Linear probing step should not have common divisor with M (size of the table) for any $k \Rightarrow$ *all* positions in the table can be reached

17.12

1.2 Choose a good hash function

What is a good hash function?

Let k be a natural number.

Hashing should give a uniform distribution of the hash values, but this depends on the distribution of the keys in the considered data set.

Example: Hashing of English words

- hash function: ASCII-value of the first letter *poor choice:* not an even distribution.

17.13

String hashing in Java

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

- For long strings: only consider a finite number of 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: high risk for a collision patterns

17.14

Ideas for hash functions

- Memory addresses
  - Interpret memory address of an object as an integer
  - Works well when using pointers as keys (difference between equality and identity).
- Interpret as integers
  - Interpret the bits in a key as an integer
  - Can work for keys with few numbers of bits
- Sum of components
  - Divide the bits in the key into components of fixed length (e.g. 16 or 32 bits) and sum the components (Ignoring overflows.)
  - Can work for numerical keys of fixed lengths with more bits than those in an integer

17.15

Possible hash functions

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

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

  - Evaluate the polynom

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

    for some fixed value z. (Ignore overflows)
  - Works well for hashing strings . (e.g. $z = 33$ gives at most 6 collisions for 50000 English words.)
- Polynom $p(z)$ can be evaluated in $O(n)$ steps with Horner's evaluation:
  - Iterative evaluation. Each polynom can be evaluated in $O(1)$ steps based on the previous polynom in the sequence

$$p_0(z) = a_{n-1}$$
$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \; (i = 1, 2, \ldots, n-1)$$

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

17.16

String hashing in Java

`hashCode()` for `String` in Java

```
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

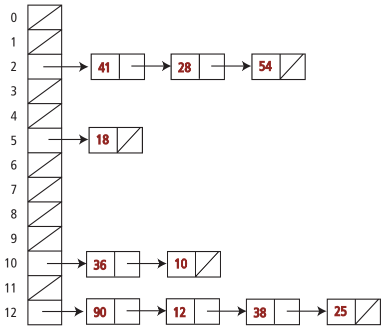
char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

17.17

Algorithmic complexity attacks

Is the assumption on the uniform distribution of the keys important in practice?

- In critical applications you want to avoid timing "surprises"
- An attacker could craft inputs or packages to produce a hash-collision based DOS-attack [Crosby-Wallach 2003]
- Regular expression denial of service [Staicu-Pradel-2018]



17.18

2 Tree-based applications

2.1 Text compression

Text compression

Greedy algorithms: algorithms that solve a piece of the problem at a time. Each step performs the best "local" actions.

- The greedy approach is a general paradigm when designing algorithms, it builds on the following:
  - Configurations: different choices, sets or values to find
  - objective function: Configurations are assigned a score to be maximized or minimized
- The approach works best for problems that enjoy the *greedy-choice*-property:
  - a globally optimal solution can always be found via a series of local improvements starting from a configuraiton

for many problems, the greedy approach does not give an optimal solution but good approximations. 17.19

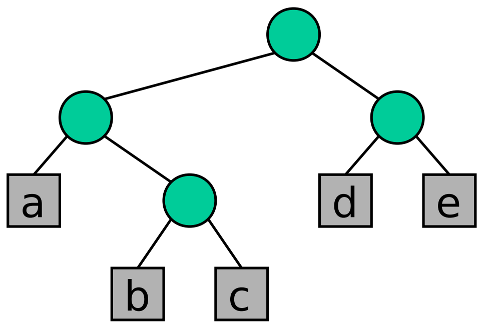
Text compression

- Given a string X, encode X as a shorter string Y
  - Saves memory/bandwidth
- A good way to do it: *Huffman encoding*
  - Compute the frequency $f(c)$ of each character c
  - Use short codes for frequent characters
  - No code is a prefix of another code
  - Use an optimal coding tree to determine the codes 17.20

An encoding tree example

- A code maps each character of an aplhabet to a binary code
- A prefix-code is a binary code ensures no code word is prefix of another code word
- An encoding tree represents a prefix-code
  - Each external node stores a character
  - The code-word for a character is given by the path from the root to the external node that stores the character (0 for the left child and 1 for the righ child)

00	010	011	10	11
a	b	c	d	e



17.21

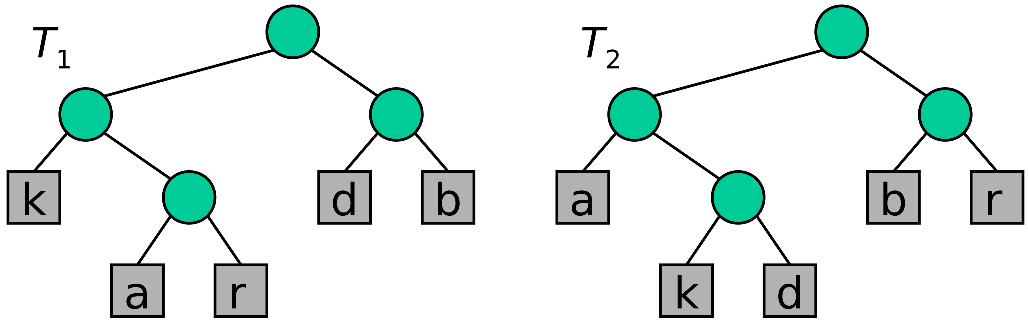
Optimization of encoding trees

- Given a string X, we want to find a prefix-encoding for the characters in X that gives a short encoding of X
  - Common characters should get short code-words
  - Unusual characters can get get longer code-words

Exampel: $X = abrakadabra$

- T_1 encodes X with 29 bits
- T_2 encodes X with 24 bits

a	b	k	d	r
5	2	1	1	2



17.22

Huffman's algorithm

- Given a string X, Huffman's algorithm constructs a prefix-encoding that minimizes the size of the resulting encoding of X
- The algorithm runs in $O(n + d \log d)$ time complexity, where n is the size of X and d is the number of distinct charachters in X
- A heap based priority queue is used as an extra data-structure

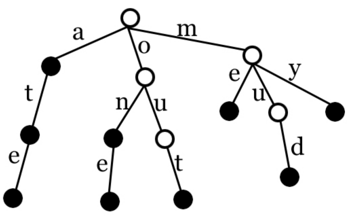
function $\text{HUFFMANENCODING}(X, |X| = n)$
  $C \leftarrow \text{DISTINCTCHARACTERS}(X)$
  $\text{COMPUTEFREQUENCIES}(C, X)$
  $Q \leftarrow$ new empty heap
  for all $c \in C$ **do**
    $T \leftarrow$ new single node tree to encode c
    $Q.\text{INSERT}(\text{GETFREQUENCY}(c), T)$
  while $Q.\text{SIZE}() > 1$ **do**
    $f_1 \leftarrow Q.\text{MIN}()$
    $T_1 \leftarrow Q.\text{REMOVEMIN}()$
    $f_2 \leftarrow Q.\text{MIN}()$
    $T_2 \leftarrow Q.\text{REMOVEMIN}()$
    $T \leftarrow \text{JOIN}(T_1, T_2)$
    $Q.\text{INSERT}(f_1 + f_2, T)$
  return $Q.\text{REMOVEMIN}()$

17.23

2.2 Prefix-trees

Prefix-trees (Trie)

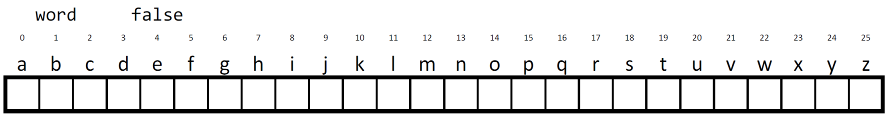
- trie: An ordered tree datastructure used to store a data set, usually strings, an optimized to perform prefix-searches
  - Example: Are there words in the set that start with the prefix `mart`?
  - Lexicon-class in labb5 uses such a datastructure
  - Idea: instead of a binary tree, use a "26-ary" tree
    - Each node may have 26 children: one for each letter A-Z
    - insert a word in the trie by following the suitable pointers associated to the children



17.24

Trie-node

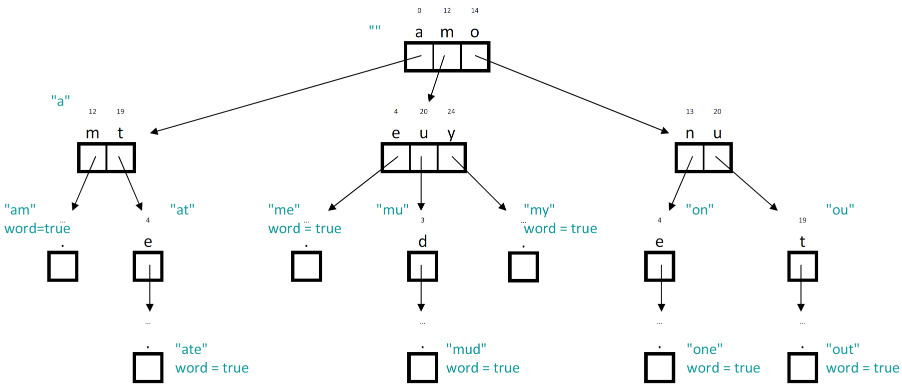
```
struct TrieNode {
   bool word;
   TrieNode* children[26];

   TrieNode() {
      this->word = false;
      for (int i = 0; i < 26; i++) {
         this->children[i] = nullptr;
      }
   }
};
```



17.25

Trie with data

- After having inserted "am", "ate", "me", "mud", "my", "one", "out":



17.26

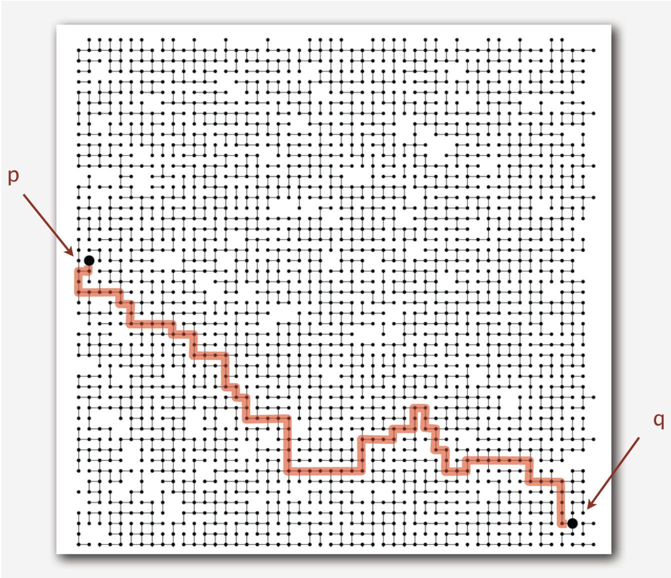
2.3 Union/Find

Partitions with Union/Find-operations

- makeSet(x): create a singleton containing x and returns the position where x is stored
- union(A,B): returns the set $A \cup B$, consumes the old A and B.
- find(p): returns the set that contains the element at position p.

17.27

Example: Connectivity



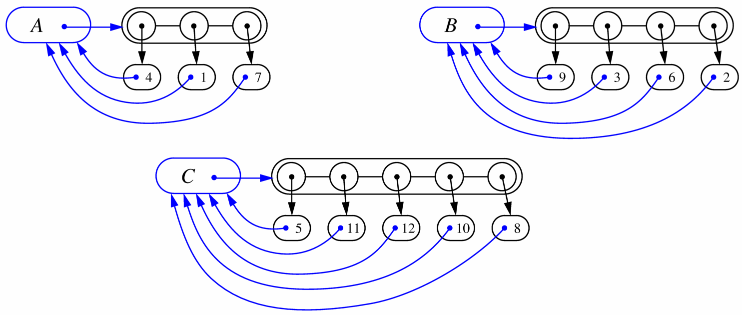
Quesiton: is there a path between p and q?

- Pixels in a digital photo
- Computers in a network
- Friends in a social media
- Transistors in a chip
- Elements in a mathematical set
- Variable names in a computer program

17.28

List based implementation

- Each set is stored as a sequence captured by a linked list
- Each node contains an element and a reference to the set name



17.29

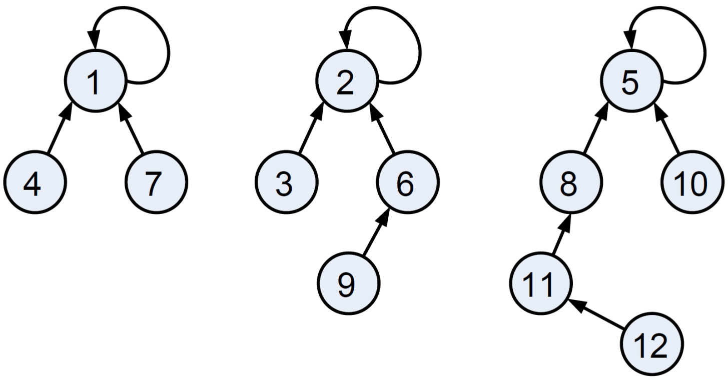
Analysis of the list based representation

- When creating unions, always move elements from the smaller set to the larger set
  - Each time an element is moved, it will be a member of a set that is at least twice as large as the old set
  - Hence, an element can be moved a maximum of $O(\log n)$ times
- Total time to perform n union and find-operations is $O(n \log n)$

17.30

Tree based implementation

- Each element is saved in a node that contains a pointer to a set name
- A node v that points to itself is also a set name
- Each set is captured with a tree with a self-pointing node as a root
- ex. sets ”1”, ”2” och ”5”:

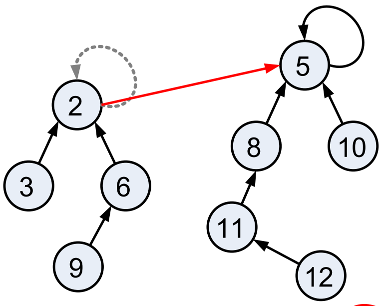


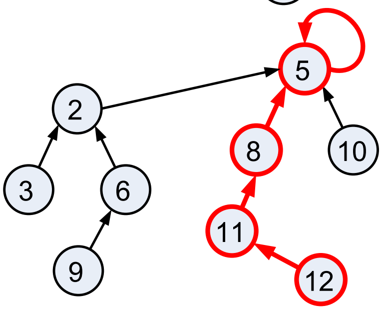
17.31

Operations

- To perform a union, just let the root of the tree point to the root of the other tree

- To perform a find, follow the pointer from the given node to the self-pointing one



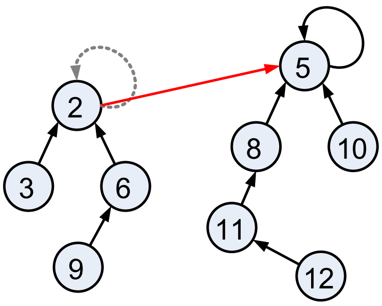


17.32

A heuristic

- Union by sizes:
  - When performing a union, let the root of the smaller tree point to the root of the larger one

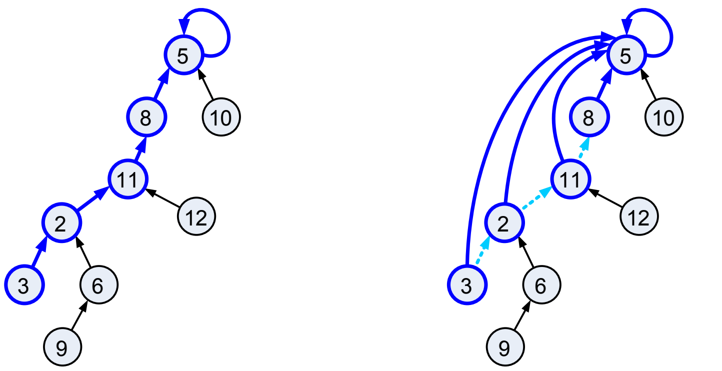
- Results in $O(n \log n)$ steps to perform n unions and find operations:
  - Each time we follow a pointer, we get to a tree that is at least twice the size of the previous subtree
  - Hence, we will follow at most $O(\log n)$ pointers during find



17.33

One more heuristic

- Path compression:
  - After find is executed, make all nodes on the path point to the root

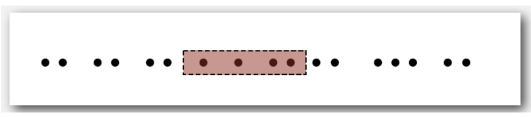


17.34

2.4 Geometric search

One dimension range search

- Extending ordered symbol tables
  - insert key-value pairs
  - search for key k
  - Range seach: find all keys between k_1 and k_2
  - Range size: the number of keys between k_1 and k_2
- Applications
  - Database queries
- Geometric interpretation:
  - Keys are points on a line
  - Find/count the number of points in a given range

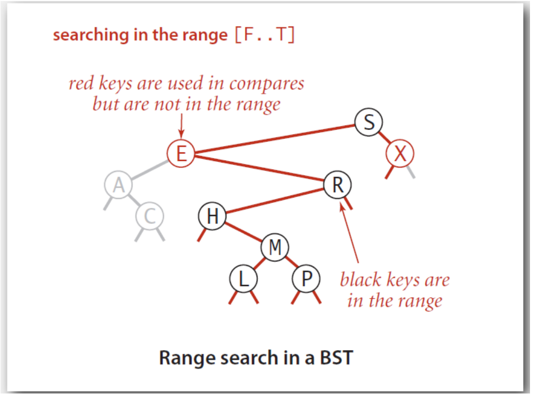


insert B	B
insert D	B D
insert A	A B D
insert I	A B D I
insert H	A B D H I
insert F	A B D F H I
insert P	A B D F H I P
count G to K	2
search G to K	H I

17.35

Range search in one dimension with BSTs

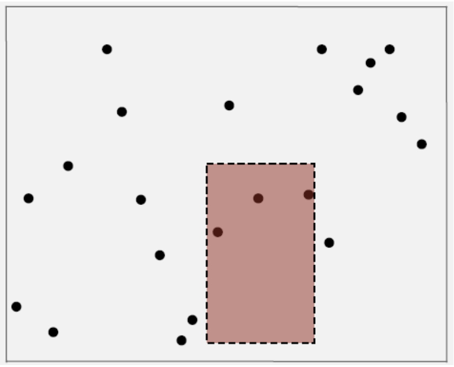
- Find all keys between k_1 and k_2
  - Find, recursively, all keys in the left subtree (if any can be in the range)
  - Control key in current node
  - Find, recursively, all keys in the right subtree (if any can be in the range)



17.36

Two dimensions range search

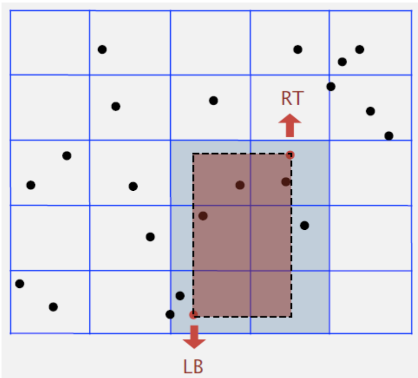
- Extending ordered symbole tabkes ti 2D-keys
  - insert a 2D-key
  - Search for a 2D-key
  - Range search: find all keys in a 2D-range
  - Range size: number of keys in a 2D-range
- Applications:
  - Networks, Chip design, databases
- Geometric interpretation:
  - Keys are points in a plane
  - Find/count keys in a given rectangle



17.37

Range search in two dimensions with a grid

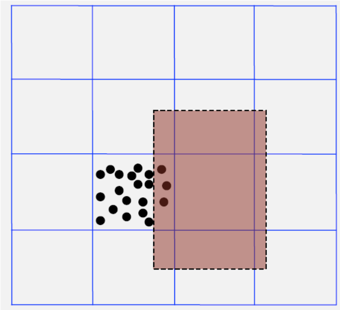
- Divide the plane into a grid with $M \times M$ squares
- Create a list of points in each square
- Use 2D-array to directly index the squares
- Range search: only examine the squares that overlap the query



17.38

Clustering

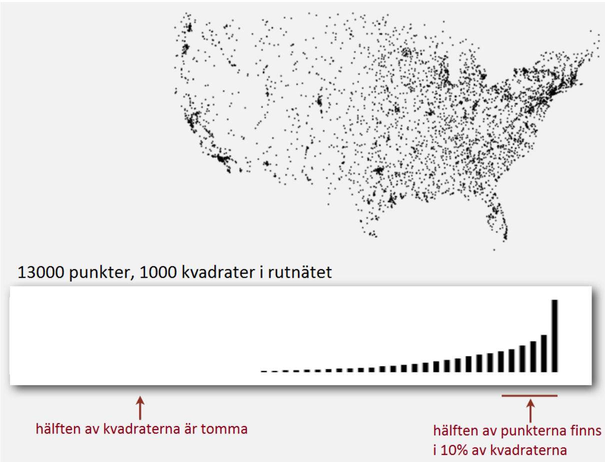
- Grid implementation:
  - Fast, simple solution for well distributed points
- Problem: Clustering is a known phenomenon for geometrical data
  - Some lists get too long, although the average length is shor
  - Need for a data-structe that *adapts* to the data



17.39

Clustering

- Grid implementation:
  - Fast, simple solution for well distributed points
- Problem: Clustering is a known phenomenon for geometrical data
  - Exempel: kartdata

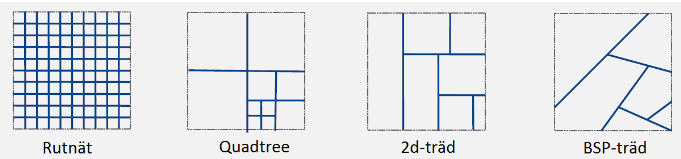


17.40

Tree structures

Use a *tree* to recursively partition the plane

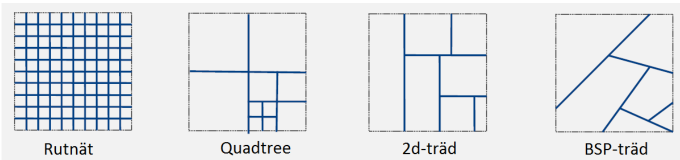
- Grid: Divide the plane uniformly into squares
- Quadtree: Divide the plane recursively into four squares
- 2D-tree: Divide the plane recursively into two half planes
- BSP-tree: (Binary Search Partition) Divide the tree recursively into two regions



17.41

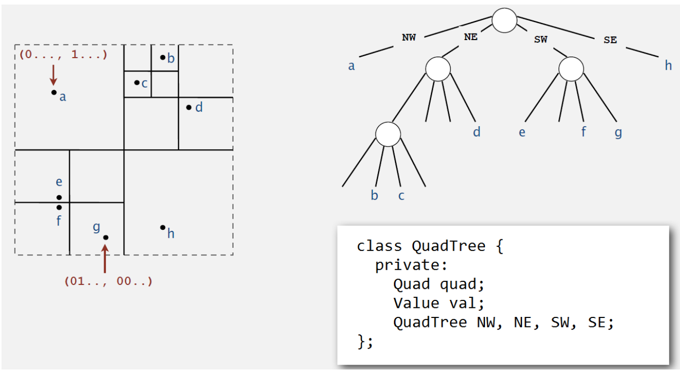
Applications

- Ray-tracing
- Range search in 2 dimensions
- Fligh simulators
- N-body simulations
- Collision detection
- Astronomical databases
- Search for closest neighbors
- Adaptative grid generation
- Remove hidden surfaces and shading



Quadtree

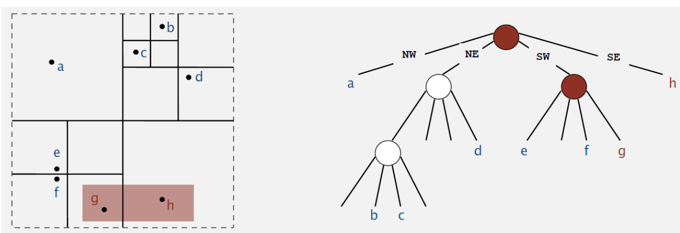
- Idea: Divide tha plane recursively into 4 squares
- Implementation: 4-way tree (actually a trie)



- Advantage: Good performance when clustered data
- Drawback: Arbitrary depth!

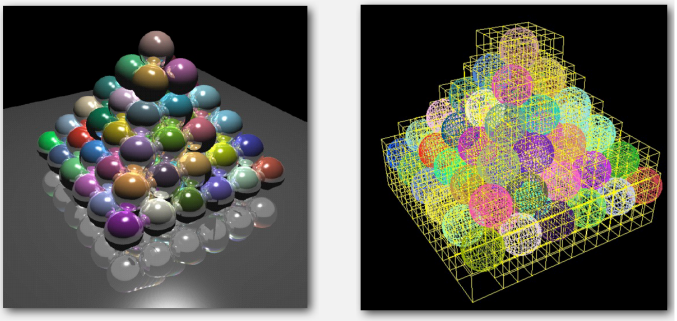
Quadtree: range search in 2D

- find recursively all keys in NE sqaure (if any can be found there)
- find recursively all keys in NW sqaure (if any can be found there)
- find recursively all keys in SE sqaure (if any can be found there)
- find recursively all keys in SW sqaure (if any can be found there)



The dimensionality problem

- Range search in k dimensions
  - Main application: Multidimensional databases
  - 3D: Octree: divide recursively the 3D space in 8 octants
  - 100D: Centree: Divide recursively the 100D space into 2^{100} centrants???

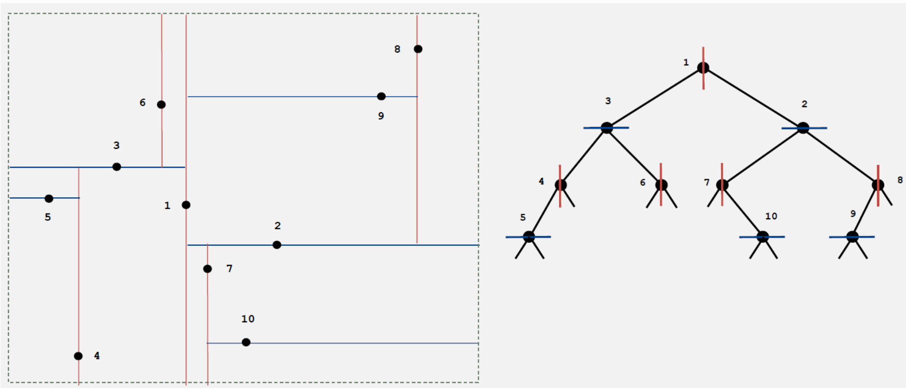


Raytracing with octrees
http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html

17.45

2D-tree

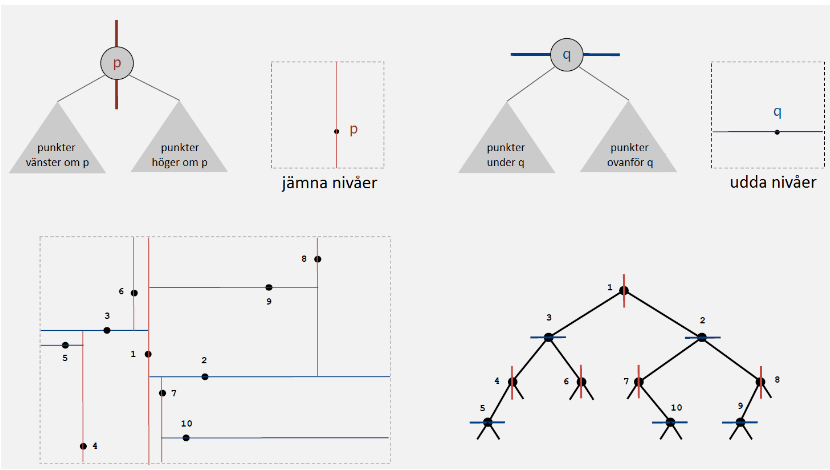
Divide recursively the plane into two half plances



17.46

2D-tree

- Data structures: BST, but alternate using x- and y-coordinates as key
  - Seach returns a rectangle containing a point
  - Insertion further divides the plane

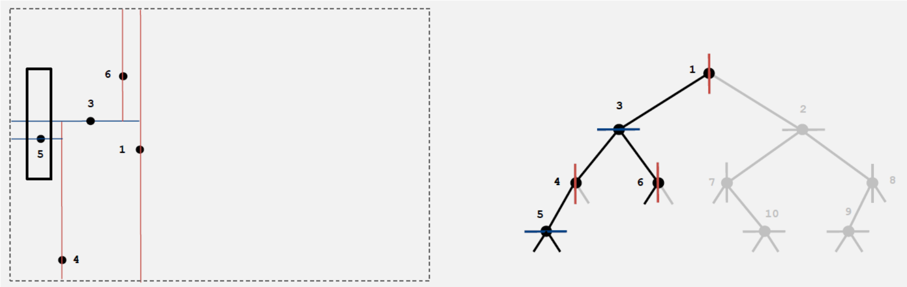


17.47

2D-tree: Range search in 2D

Find all points in a rectangle given by the query (rectangle sides are parallel with the coordinates)

- Control if current points is in the rectangle
- Recursively search in the left/top subtrees (if any can be part of the rectangle)
- Recursively search in the righ/lower subtrees (if any can be part of the rectangle)

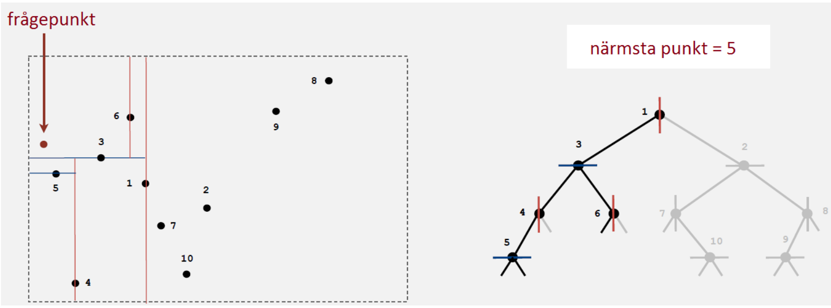


17.48

2D-tree: Search for closest neighbor

Find a point that is closest to a given point

- Control distance from current point to the point in the query
- Search recursively in the left/top subtree (if they can contain a closer point)
- Search recursively in the right/lower subtree (if they can contain a closer point)
- Set up the recurisve search so that it starts looking for the point in the query

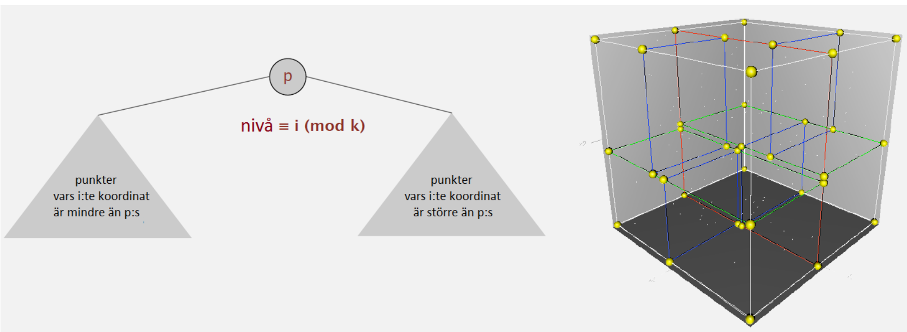


- Typical execution time: $\log N$
- Worst case (even if the tree is balanced): N

17.49

KD-trees

- KD-tree: Recursively partition the k-dimensional space in two half spaces
  - Implementation: BST, but cycle through the dimensions like in a 2D-tree



- Efficient, simple datastructure to manage k-dimensional data
  - Wide usage
  - Adapts well to high dimensional and clustered data
  - Discovered by a student (Jon Bentley) in an algorithm course!

17.50