Lecture 19

Priority Queues, Heap, Trie, Union/Find, Geometric applications of BST

TDDD86: DALP

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

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

19.1

Content

Innehåll

19.2

1 Priority Queues

Priority queues

A common occurring situation:

- Waiting list (job management on multi-user computers, simulation of events)
- If a resource becomes available, select an element from the waiting list
- The choice is based on a partial/linear order:
  - the job with the highest priority will be served first,
  - each event will occur at a specific time; the events will be processed in chronological order.

19.3

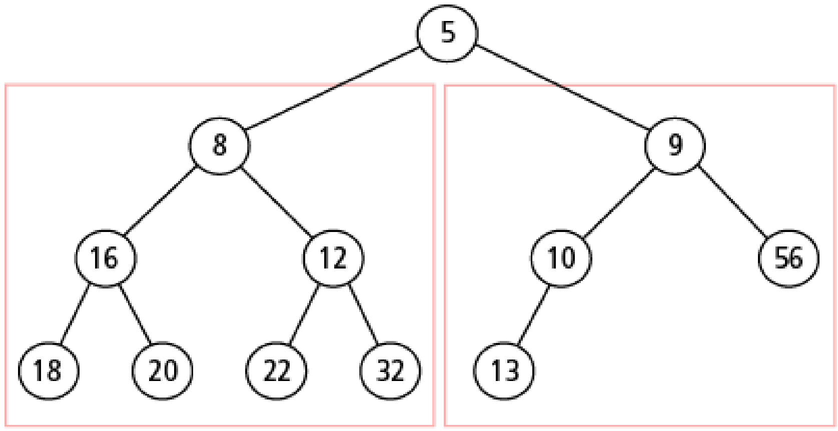
ADT priority queues

- Linearly ordered set of keys K
- We store the pairs (k, v) (as in the ADT Dictionary), *several pairs* with the *same key* are allowed
- a common operation is to retrieve couples with minimal key
- Operations on a priority queue P:
  - makeEmptyPQ()
  - isEmpty()
  - size()
  - min(): find a pair (k, v) that has the minimal k in P; returns (k, v)
  - insert(k, v): inserts (k, v) in P
  - removeMin(): removes and returns the pair (k, v) having the minimal key k; **error** if P is empty

19.4

Implementation of priority queues

- We can for example use (sorted) linked lists, BST or Skip-lists
- Another idea: use a complete binary tree where the root of each (sub) tree T contains the smallest element in T.



This is a partially ordered tree, also called "heap"! 19.5

1.1 Heaps

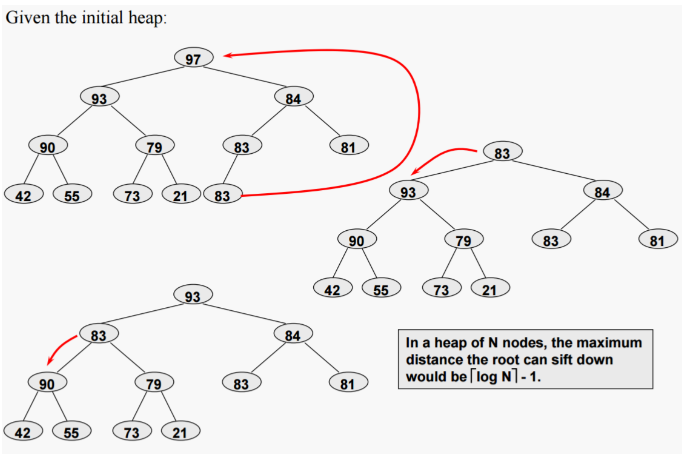
Updating a heap structure

- A heap is a complete binary tree
- With the last leaf we mean the last node in a traversal in level order

- removeMin(PQ) *// remove the root*
  - Replace the root with the last leaf
  - Reset the partial order by swapping rows below ”down-heap bubbling”

- insert(PQ, k, v)
  - Insert a new node (k, v) after the last leaf
  - Reset the partial order by ”up-heap bubbling” 19.6

Updating a heap structure



Remove the root. 19.7

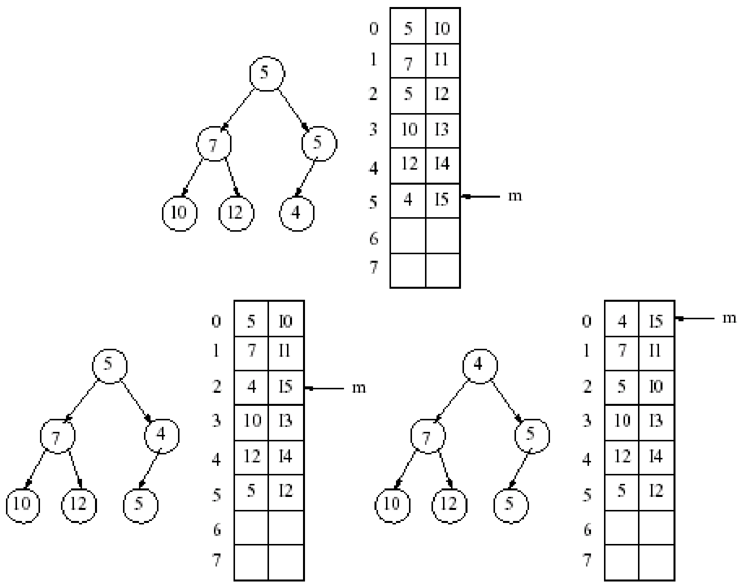
Characteristics

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

Remember the array representation of BST A complete binary tree...

- compact array representation
- ”Bubble-up” and ”bubble-down” have rapid implementations 19.8

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



19.9

Heap variants

Different partial orders

- the smallest -minimum- key is in the root (minHeap)
- the biggest key in the root (maxHeap)

Different array representations

- Numbering forward in the level orders (starting from 0 or 1)
- numbering backwards in level order (starting with 0 or 1)

19.10

1.2 Application

Greedy algorithms

Algorithms that solve a piece of the problem at a time. Each step done gives the best return and costs the least.

- The greedy method is a general paradigm for the design of algorithms based on the following:
  - configurations: different choices, collections or values to find
  - goal function: configurations are assigned a score that we will maximize or minimize
- It works well for problems with *greedy-choice* property:
  - a globally optimal solution can always be found by a series of local improvements from an initial configuration

For many problems, greedy algorithms do not provide optimal solutions but maybe decent approximate solutions. 19.11

Text compression

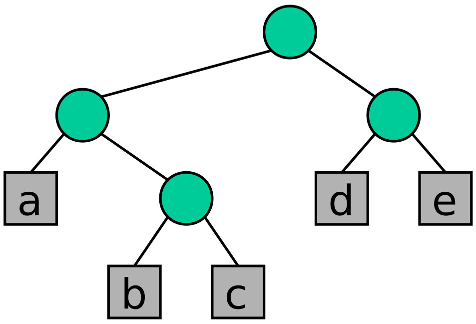
- Given a string X, encode X in a shorter string Y
  - Save memory/bandwidth
- A good way to do it: *Huffman coding*
  - Calculate the frequency $f(c)$ for each character c
  - Use short codes for characters with high frequency
  - No code word is the prefix of another code word
  - Use optimal coding tree to determine the code words

19.12

Example of coding tree

- A code maps each character in an alphabet to a binary code word
- A prefix code is a binary code such that no code word is a prefix of another code word
- A coding 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 storing that character (0 for a left child and one for a right child)

00	010	011	10	11
a	b	c	d	e



19.13

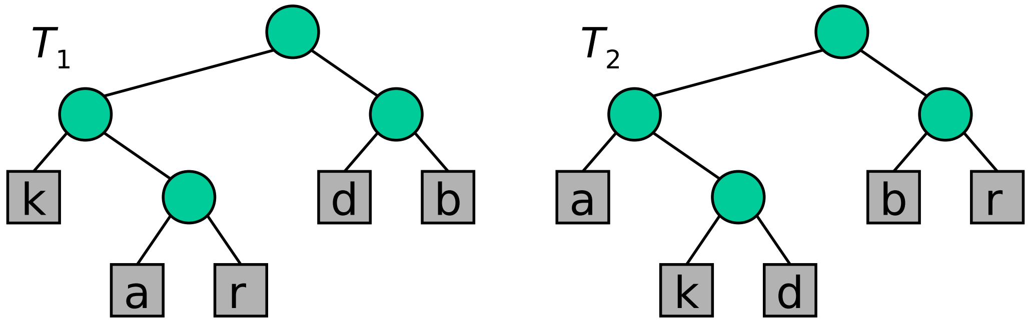
Optimization of the coding tree

- Given a text string X, we want to find a prefix code for a character in X which gives a short coding of X
  - Common characters receive short code words
  - Unusual characters should get long code words

Example: $X = abrakadabra$

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

a	b	k	d	r
5	2	1	1	2



19.14

Huffman's algorithm

- Given a string X, Huffman algorithm constructs a prefix code that minimizes the size of the encoding of X
- The algorithm runs in $O(n + d \log d)$ time, where n is the size of X and d is the number of distinguished characters in X
- A heap-based priority queue is used as an additional data structure

function HUFFMANENCODING($X, |X| = n$)
  $C \leftarrow$ DISTINCTCHARACTERS(X)
  COMPUTEFREQUENCIES(C, X)
  $Q \leftarrow$ new empty heap
  for all $c \in C$ **do**
    $T \leftarrow$ a new tree node that stores c
    Q.INSERT(GETFREQUENCY(C), 1)
  while Q.SIZE() > 1 **do**
    $f_1 \leftarrow Q$.MIN()

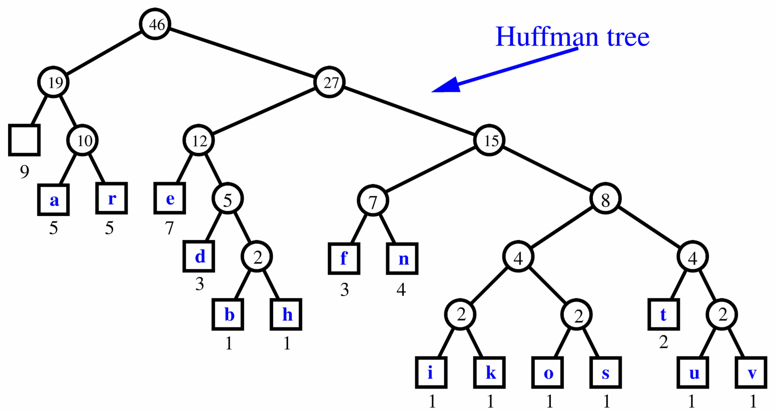
$T_1 \leftarrow Q.\text{REMOVEMIN}()$
$f_2 \leftarrow Q.\text{MIN}()$
$T_2 \leftarrow Q.\text{REMOVEMIN}()$
$T \leftarrow \text{JOIN}(T_1, T_2)$
$Q.\text{INSERT}(f_1 + f_2, T)$
return $Q.\text{REMOVEMIN}()$

19.15

Example

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

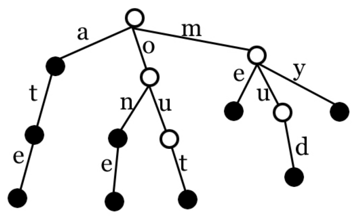


19.16

2 Trie

Trie (prefix-tree)

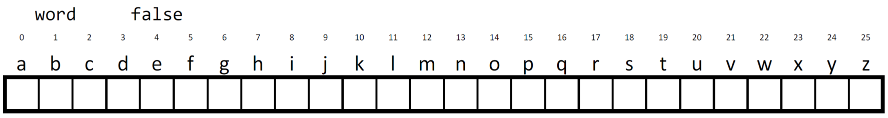
- trie: An ordered tree used to store a variety of data, usually strings, optimized to perform prefix search
  - Example: Starting a few words in the set with the prefix `mart`?
  - Dictionary-class in lab5 uses such a data structure
  - Idea: instead of a binary tree, use a “26-descendenttree
    * Each node has 26 children: one for each letter A to Z
    * add the word in a trie by following the appropriate child pointer



19.17

Trie-node

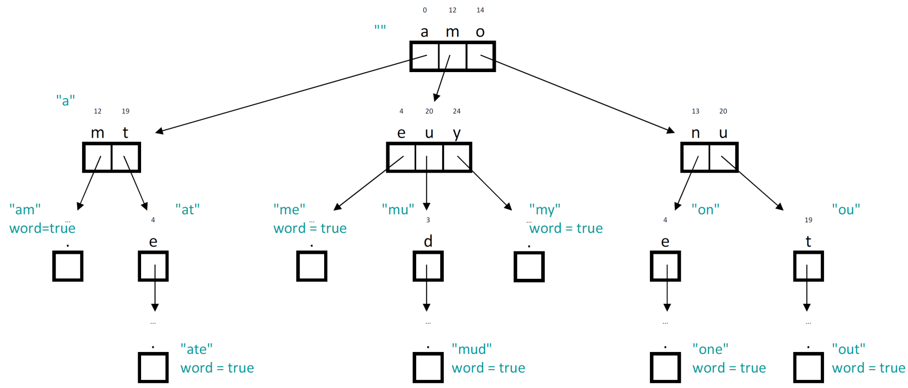
```
struct TrieNode {
   bool word;
   TrieNode* children[26];

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



19.18

Trie with data

- After inserting “am”, “ate”, “me”, “mud”, “my”, “one”, “out”:



19.19

3 Union/Find

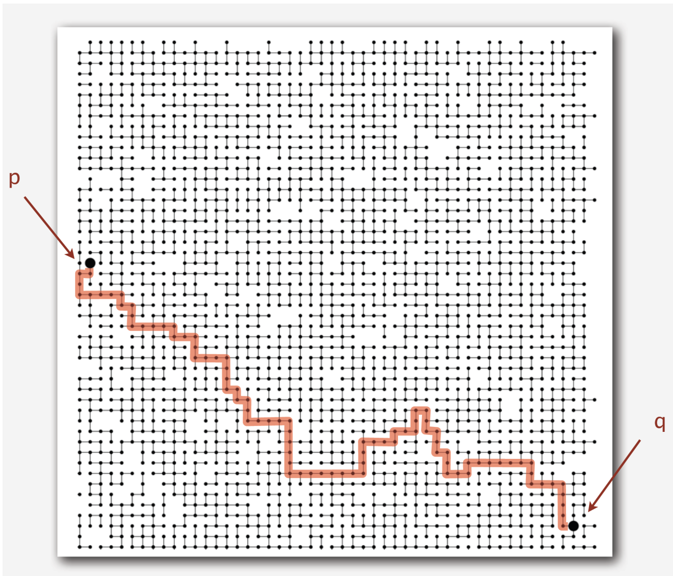
Partition Rings with Union/Find-operations

Partition rings represent a sorted list of virtual nodes, where virtual nodes are just hashes based on the actual node. Usually the sorted list just contains the hashes themselves, and a companion map is used to translate from the virtual node hash back to its actual node.

- makeSet(x): Create a set that contains only element x and returns the position that stores x.
- union(A,B): Returns the set $A \cup B$, destroy the old A and B.
- find(p): Returns the set that contains the element in position p.

19.20

Example: Dynamic connectivity



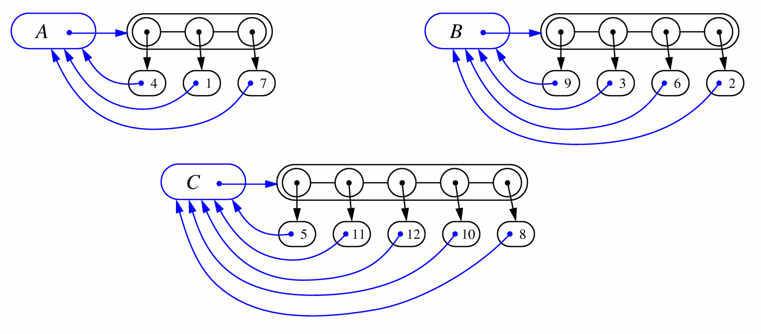
Question: is there a path between p and q?

- Pixels in a digital photo
- Computers on a network
- Friends on a social network
- Transistors in a computer chip
- Elements of a mathematical set
- Variable names in a computer program
- Metallic parts of a composite system

19.21

List-based implementation

- Each set is stored as a sequence represented by a linked list
- Each node stores an object containing an element and a reference to the set name



19.22

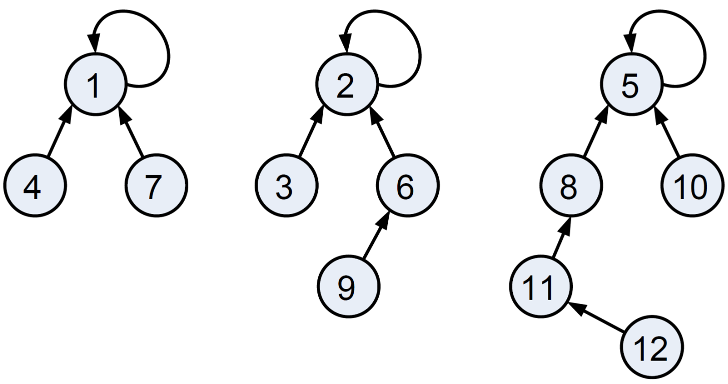
Analysis of list-based representation

- When the union is carried out, always move elements from the smaller set to the larger set
  - Each time an element is moved, it comes to a set (the new one) which is at least twice as large as the old set.
  - Thus, an element can be moved up to $O(\log n)$ times
- Total time to perform n union- and find-operations is $O(n \log n)$

19.23

Tree-based implementation

- Each element is stored in a node that contains a pointer to a set name
- A node v whose the pointer points to node v is also a set name
- Each set is a tree rooted in a node with self-referenced set name pointer
- E.g. the sets ”1”, ”2” and ”5”:

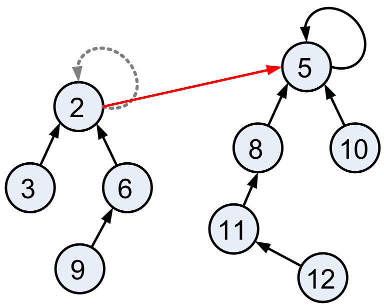


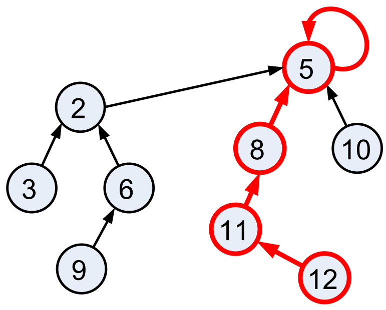
19.24

Operations

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

- To perform find, follow the set name pointers from the start node to a self-referenced node!



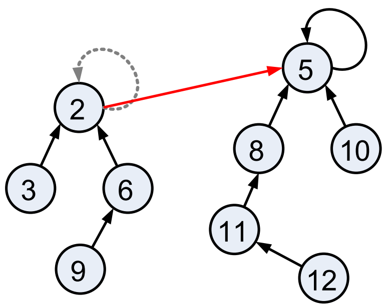


19.25

A heuristic

- Union via size:
  - When the union is carried out, the root of the smaller tree points to the root of the larger one

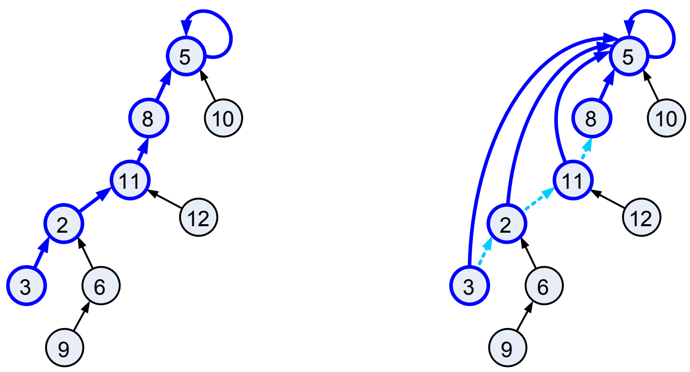
- $O(n \log n)$ time to perform n union- and find-operations:
  - Every time we follow a pointer, we come to a subtree that is at least twice as large as the previous subtree
  - Thus, we end up by following at most $O(\log n)$ pointers for any find.



19.26

Again, a heuristic

- Path compression:
  - After find"is done, compress all the pointers on the path just traversed so that they all point to the root



- $O(n \log^* n)$ time to perform n union- and find-operations.

19.27

4 Geometric search

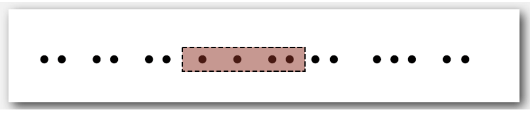
4.1 Range search

Range search in a dimension

- Extension of ordered symbol tables
  - Insert key-value pair

– Search key k
– Range search: find all keys between k_1 and k_2
– Range size: the number of keys between k_1 and k_2

- Application:
  – Database questions
- Geometric interpretation:
  – The keys are points on a line
  – Find/count points on a given range

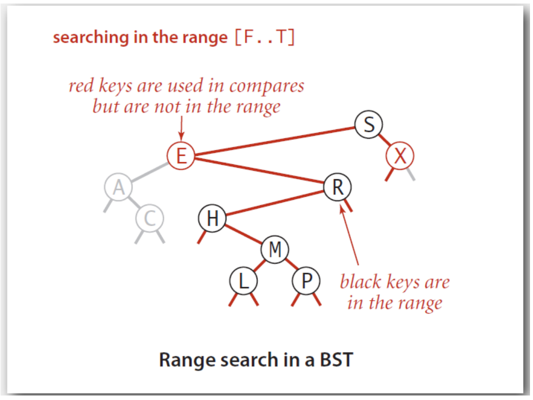


insert B	B
insert D	B D
insert A	A B D
insert I	A B D I
insert H	A B D H I
insert F	A B D F H I
insert P	A B D F H I P
count G to K	2
search G to K	H I

19.28

Range search in a dimension with BST

- Find all keys between k_1 and k_2
  – Find recursively all the keys in the left subtree (some may be in the range)
  – Check the key in the current node
  – Find recursively all the keys in the right subtree (some may be in the range)



Range search in a BST

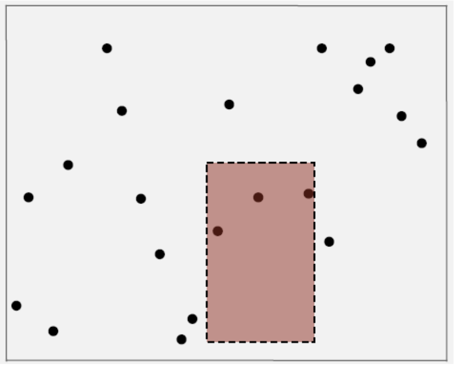
Driving time is proportional to $R + \log N$

19.29

Range search in 2 dimensions

- Extension of ordered symbol table to 2d keys
  – Insert a 2d-key
  – Search the 2d-key
  – Range search: find all keys in a 2d-range
  – Range size: the number of keys in a 2d-range
- Applications:
  – Network, circuit design, databases
- Geometric interpretation:
  – The keys are points in the plane (2d)

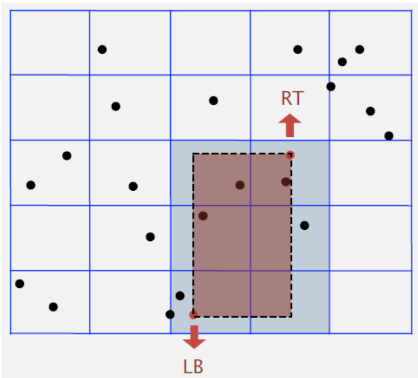
- Find/count points in a given rectangle



19.30

Range search in two dimensions with grid

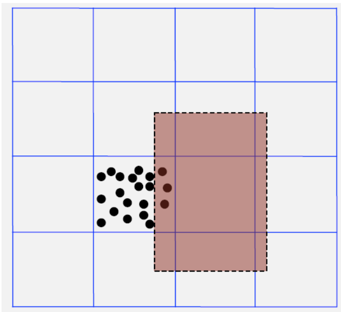
- Divide up the plane in $M \times M$-grid of squares
- Create list of points in each square
- Use 2d-array to directly index the relevant squares
- Interval search: check only the squares that overlap the issue



19.31

Clustering

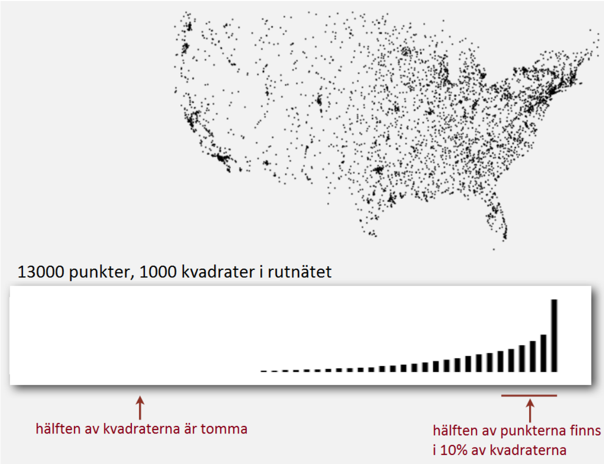
- Grid implementation:
  - Fast, easy solution for well-distributed point sets
- Problem: Clustering a well-known problematic phenomenon of geometric data
  - The lists are too long, even though the average length is short
  - Need data structure that *adapts* to data



19.32

Clustering

- Grid implementation:
  - Fast, easy solution for well-distributed point sets
- Problem: Clustering a well-known problematic phenomenon of geometric data
  - For example, map data



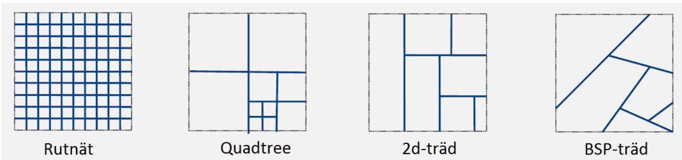
19.33

4.2 Tree structures

Tree structures

Use a *tree* to recursively split the (2d-surface) plane

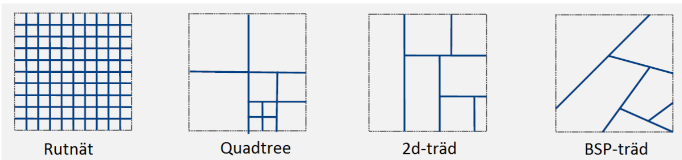
- Grid: Divide the plane uniformly in squares
- Quadtree: Divide the plane recursively into four quadrants
- 2d-tree: Divide the plane recursively into 2 half-plane
- BSP-tree: Divide the plane recursively into 2 regions



19.34

Applications

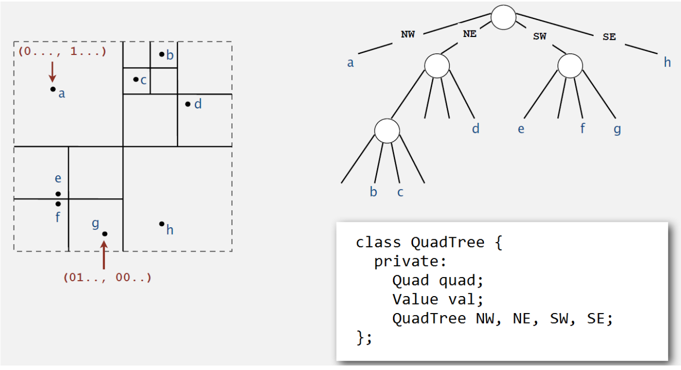
- Ray-tracing
- Range search in 2 dimensions
- flight simulators
- collision detection
- Astronomical databases
- Search for the nearest neighbors
- Adaptive grid generation
- Accelerate rendering of Doom
- Remove hidden surfaces and shading



19.35

Quadtree

- Idea: Divide the plane recursively into four quadrants
- Implementation: 4-way tree (actually a trie)

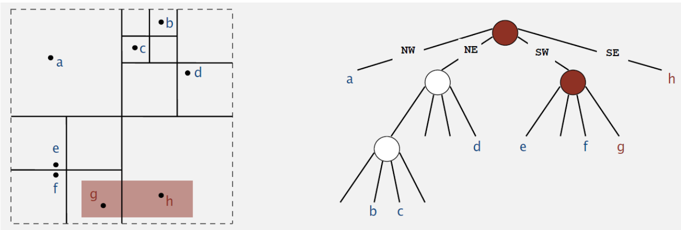


- Advantage: Good performance for clustering data
- Disadvantage: Arbitrary deep!

19.36

Quadtree: Range search in 2 dimensions

- Find recursively all the keys in the NE-quadrant (some may be in the range)
- Find recursively all the keys in the NW-quadrant (some may be in the range)
- Find recursively all the keys in the SE-quadrant (some may be in the range)
- Find recursively all the keys in the SW-quadrant (some may be in the range)

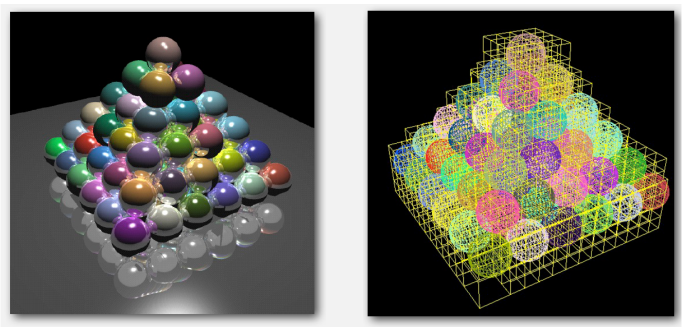


- Typical execution time: $R + \log N$

19.37

Dimensionality problem

- Range search in k dimensions
  - Main application: Multi-dimensional databases
  - 3d: Octree: recursively split up the 3D space in 8 oktanter
  - 100d: Centree: split up recuresively a 100d-space into 2^{100}?

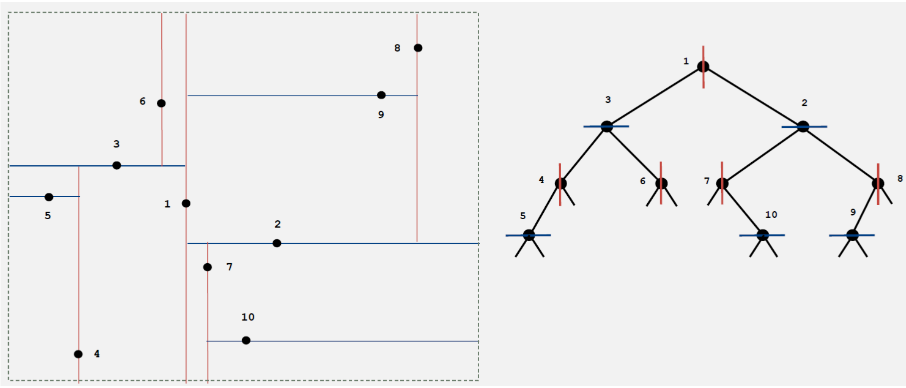


Raytracing with octrees
http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html

19.38

2d-tree

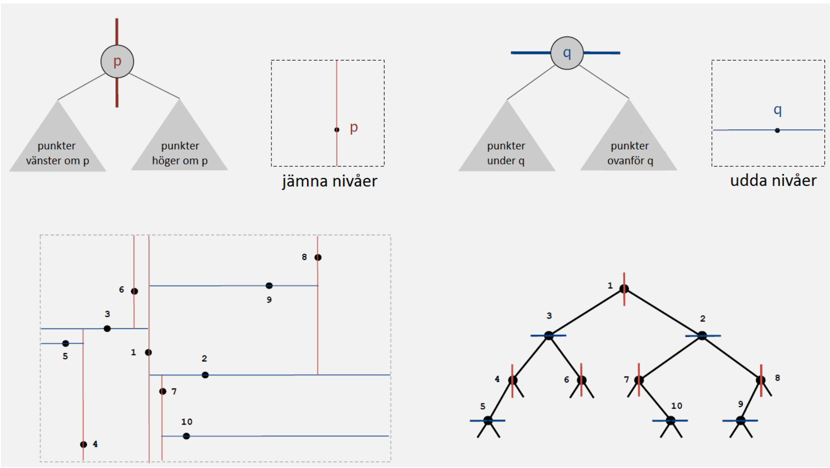
Split up recursively the plane in 2 half-planes



19.39

2d-tree

- Data structure: BST, but uses x- and y-coordinates as key
  - Search provides a rectangle containing point
  - Insertion under sub-parts further

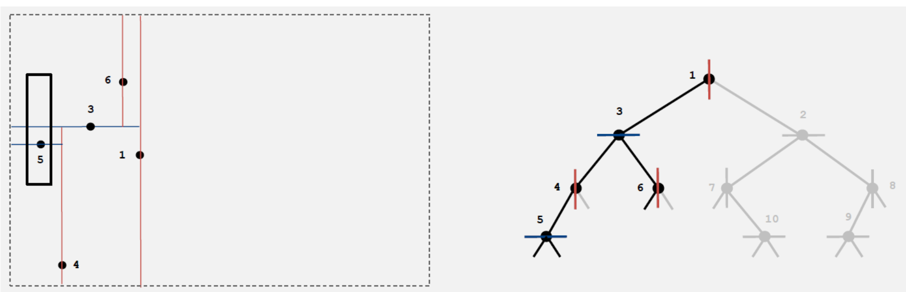


19.40

2d-tree: Range search in 2 dimensions

Find all the points of the rectangle in question (in line with the coordinate axes)

- Check the points in node located in the given rectangle
- Search recursively in the left/upper subdivision (a few points can be found in the rectangle)
- Search recursively in the right/lower subdivision (a few points can be found in the rectangle)



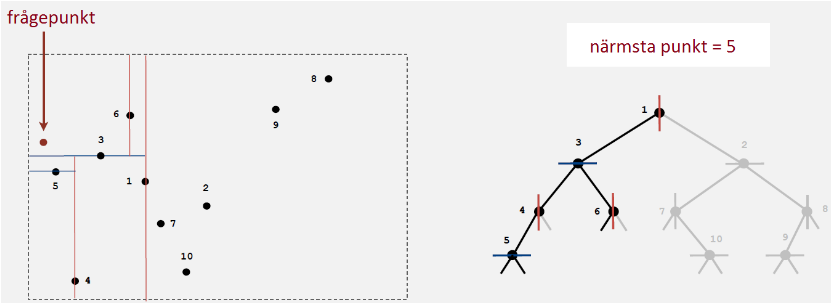
- Typical execution time: $R + \log N$
- Worst case (assuming the tree is balanced): $R + \sqrt{N}$

19.41

2d-tree: Search for nearest neighbor

Find the point closest to a given point

- Check the distance from the point in a node to the point in question
- Search recursively in the left/upper subdivision (that can contain nearer points)
- Search recursively in the right/lower subdivision (that can contain nearer points)
- Organize a recursive method so that it begins by searching for the query point

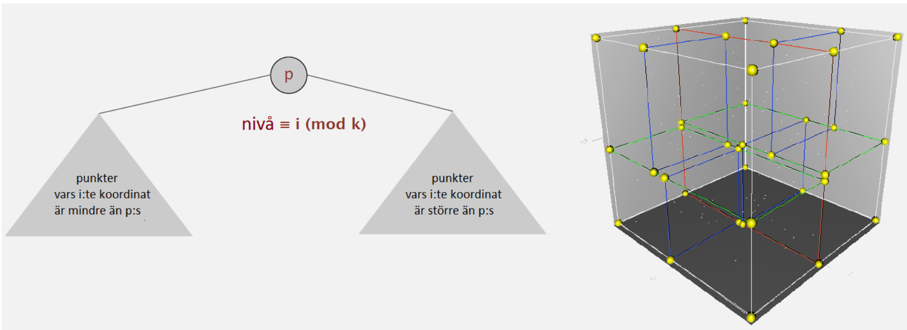


- Typical execution time: $\log N$
- Worst case (even if the tree is balanced): N

19.42

Kd-tree

- Kd-tree: Partition recuresively the k-dimensional space into two half spaces
  - Implementation: BST, but the cycling dimensions like 2d-trees



- Efficient, simple data structure to treat k-dimensional data
  - wide use
  - Adapts well to higher dimensional clustering and data
  - Discovered by a student (Jon Bentley) in an algorithm course!

19.43