Data Structures for Computer Graphics

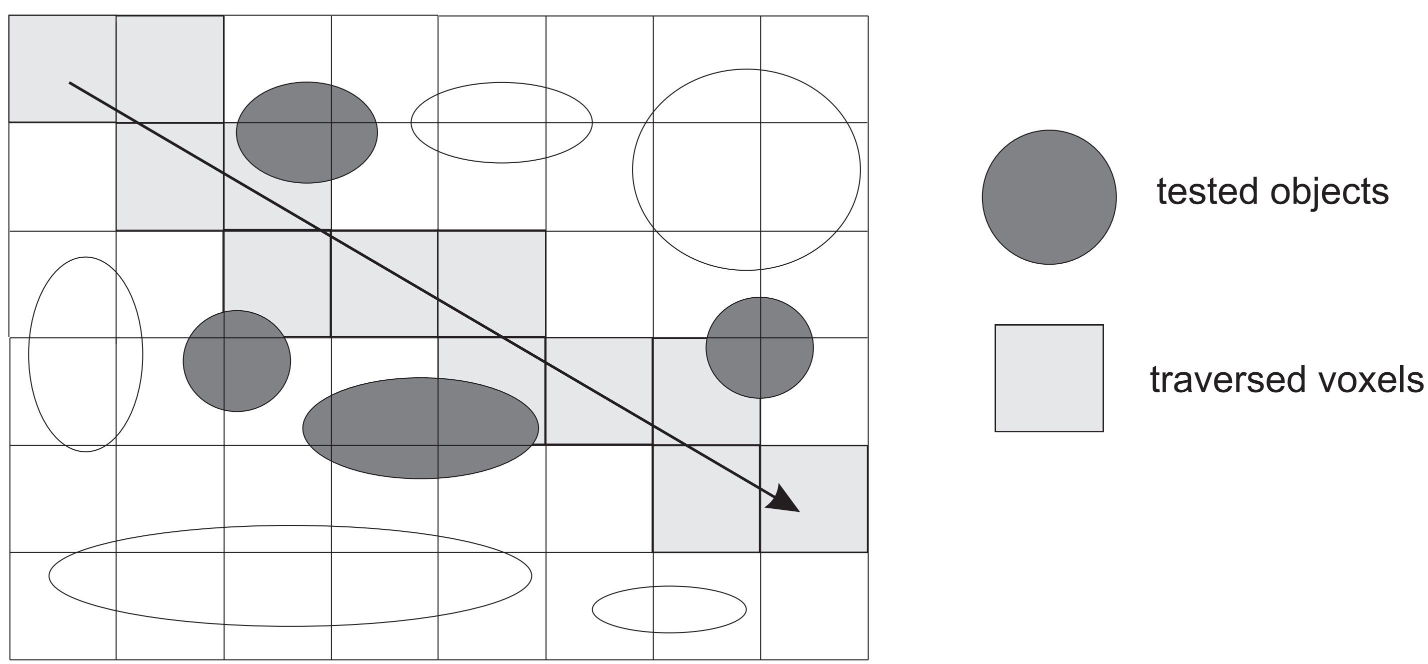
Introduction to Regular and Hierarchical Data Structures

Lectured by Vlastimil Havran

Regular Data Structures (RegDS)

- RegDS contain some regular element, such as the bucket of the same size, repeated many times.
- The number of elements is often proportional to the number of input data. (Order O(N)).
- Data dimensionality either in 1D (array), 2D, 3D, 4D, etc.
- The search sometimes possible in O(1) time.
- They perform for uniform data distributions very well.
- They do not perform well with the data that exhibit skewed (=non-uniform) distributions.

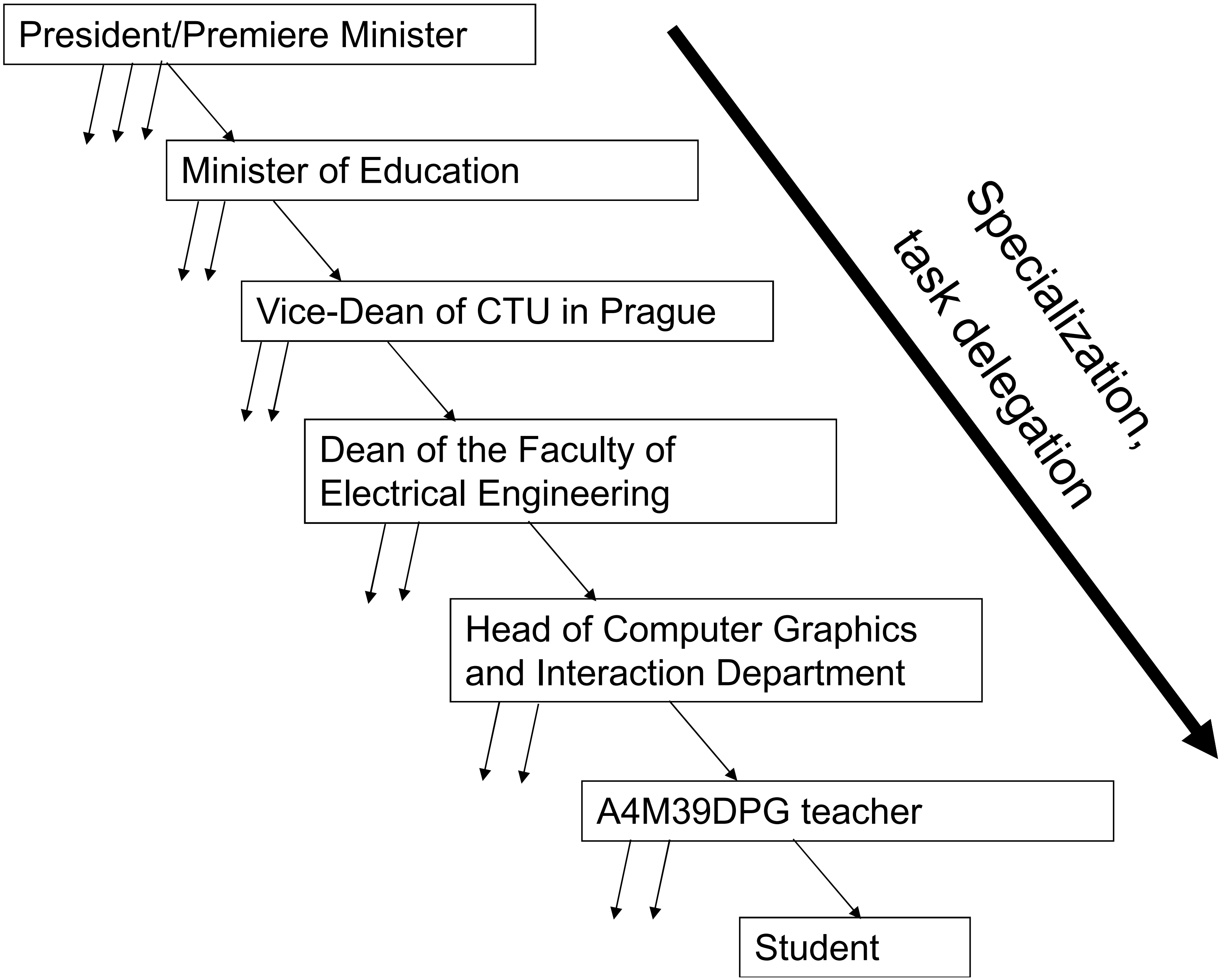
RegDS Example:
Ray Tracing with Uniform Grids



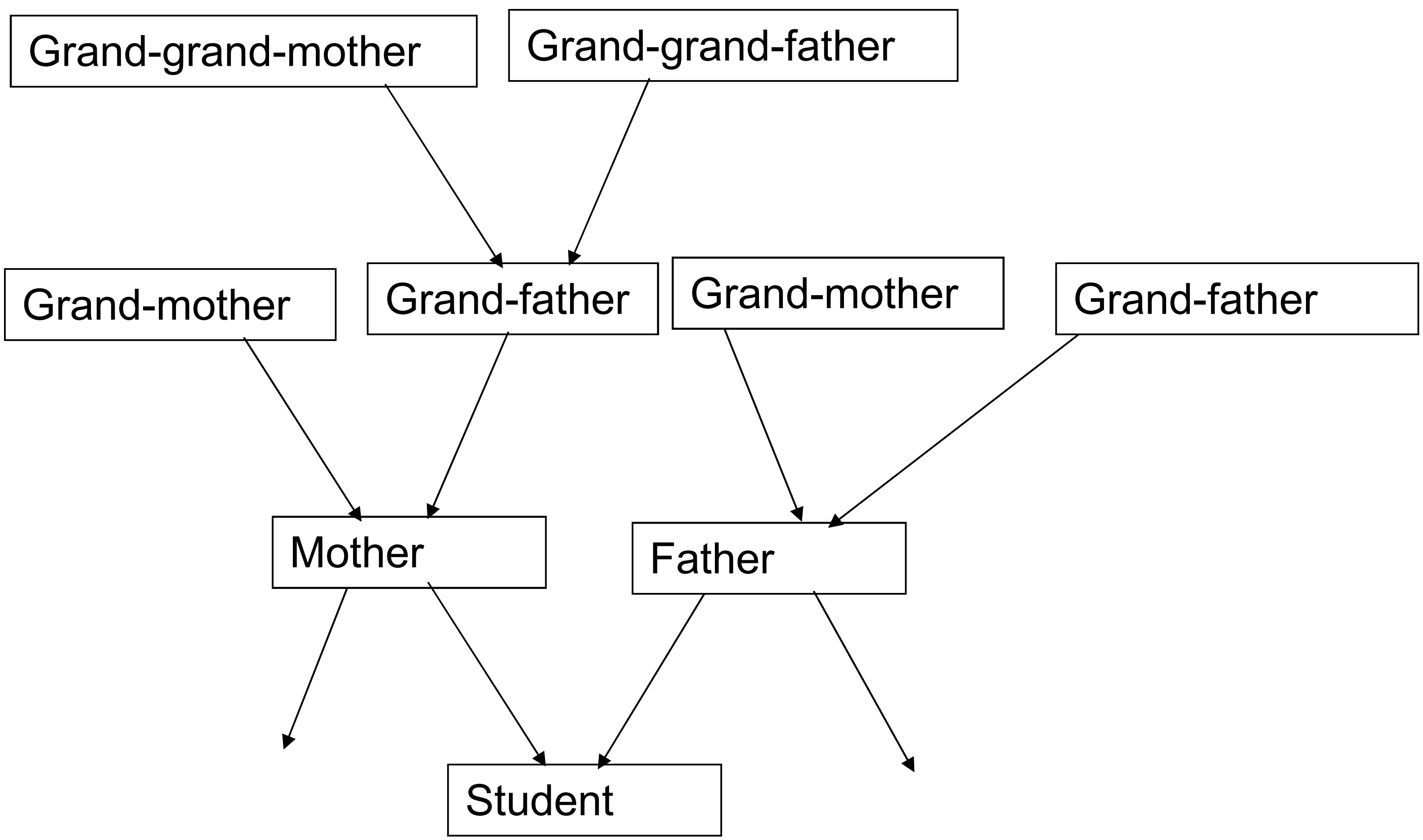
Hierarchical Data Structures (HDS)

- Why do we need a hierarchy ?
- Common concept used by mankind (in Latin *Divide et Impera*): Divide and Conquer
- Hierarchy enables delegating tasks from a single subject to several subordinate subjects. If there is nobody to delegate the task, you have to do it yourself.
- Efficient means to master the complexity and specialization required by complex tasks.

Hierarchy Example (simplified)



Hierarchy Example 2: Biological Pedigree



In Real Life

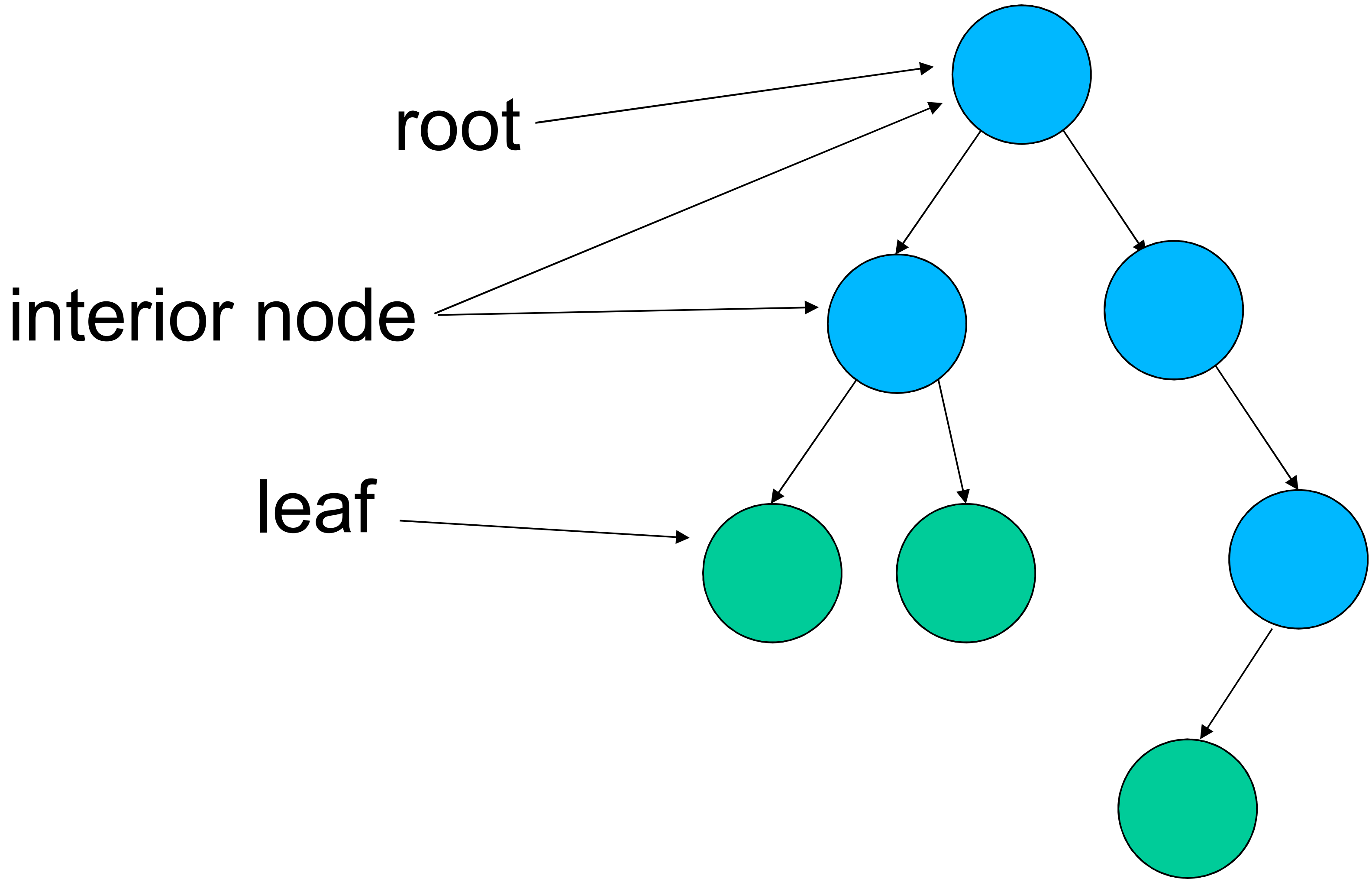
- You are the part of many hierarchies in your life, be aware of that !
- During the life people move in the hierarchies up and down.
- Note: some levels can be sometimes skipped – if you have good luck.

In Computer Graphics Applications

- Similar problems and solutions

Hierarchical Representation

= tree or even a graph (DAG)



Hierarchical Data Structures (HDS)

- Connection to sorting
- Classification
- Bounding volume hierarchies (BVH)
- Spatial subdivisions
- Hybrid data structures
- HDS construction algorithm
- Searching algorithm with HDS
- HDS for dynamic data

Connection to Sorting

Hierarchical Data Structures =

implementation of (spatial) sorting

Why ?

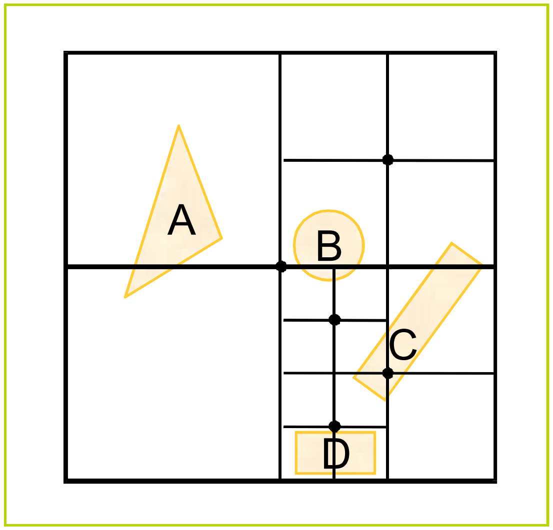
- Time complexity is O(N log N)
- Space complexity is O(N)
- For 1D hierarchy over "points" the HDS construction is clearly equivalent to quicksort

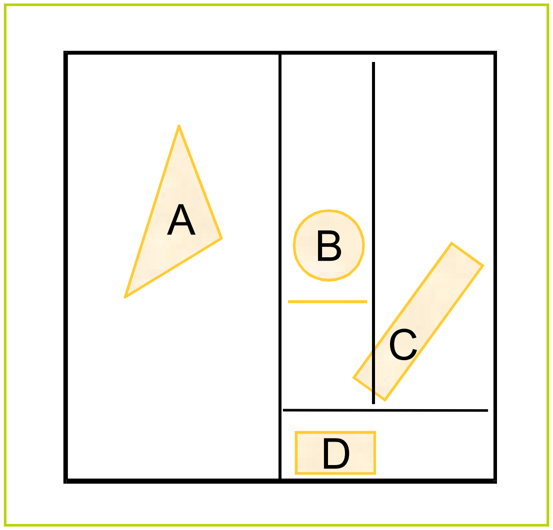
Recall Quicksort

- Pick up a pivot Q
- Organize the data into two subarrays: the left part smaller than pivot Q, the right part larger or equal than pivot Q
- Recurse in both subarrays

Examples of HDS in 2D/3D

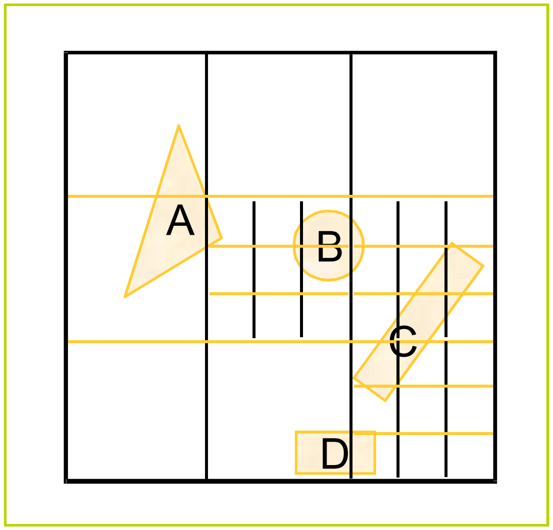
quadtree
(octree)

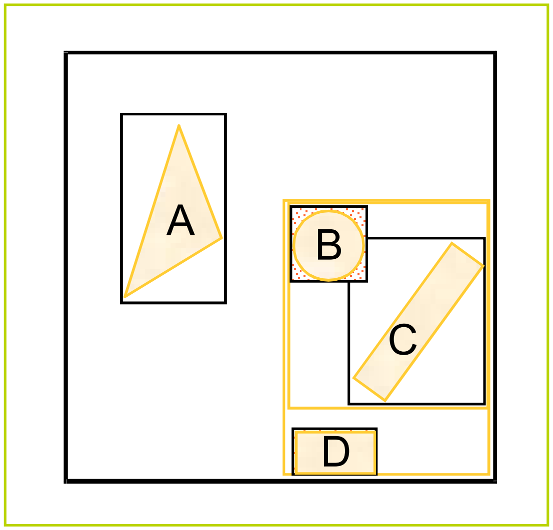




kd-tree

hierarchy
of grids





bounding
volume
hierarchy

Note that pivot is selected differently to 1D quicksort !

HDS Classification

- Data domain organization
- Dimensionality
- Data layout

HDS Classification

1) Data domain organization of HDS

- Spatial subdivisions – primarily organizing space (non-overlapping regions)
- Object hierarchies – primarily organizing objects (possibly overlapping regions)
- Hybrid data structures – spatial subdivisions mixed with object hierarchies
- (Transformations and mappings)

HDS Classification

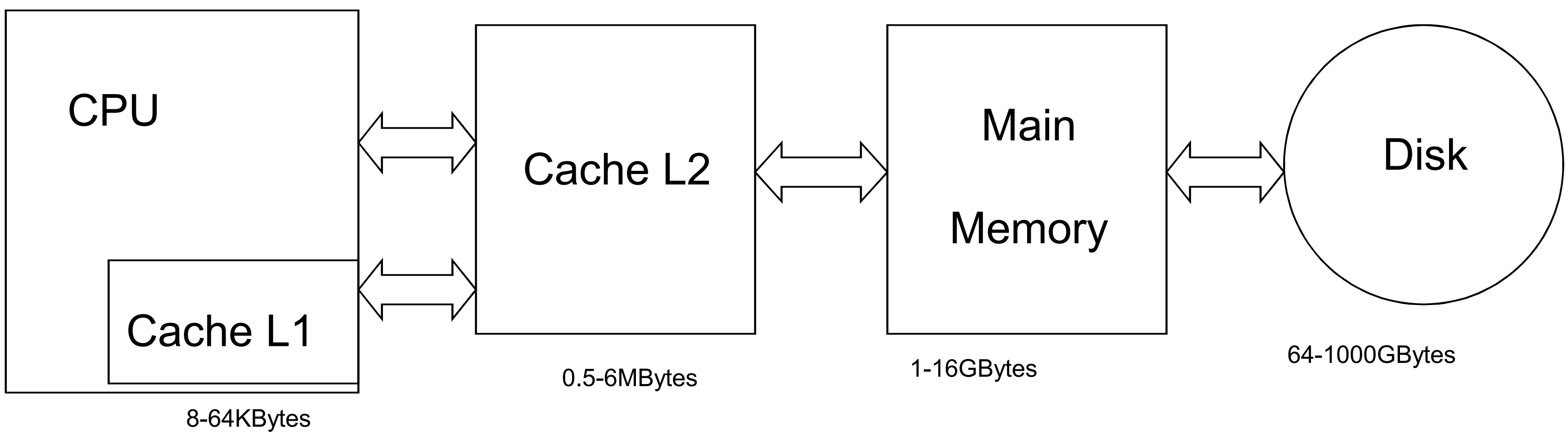
2) Dimensionality of HDS

- Necessary to represent data entities: 1D, 2D, 3D, 4D, or 5D
- Data entities: points, lines, oriented half-lines, disks, oriented hemispheres, etc.
- Possibility to extend many problems to time domain (so plus one dimension)

HDS Classification

3) HDS data layout

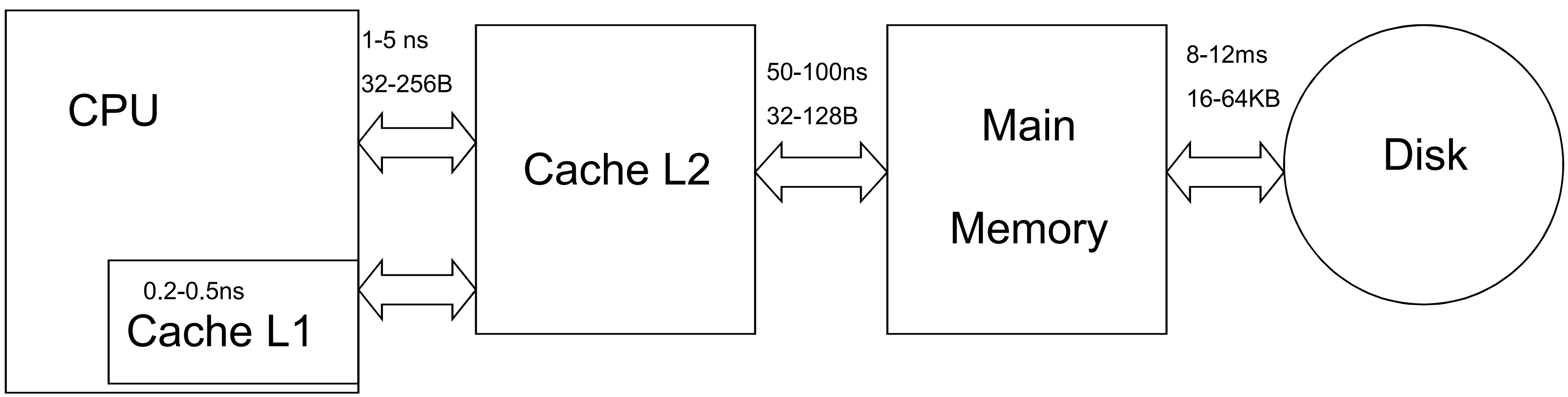
Motivation: Memory Hierarchy =
Spatial and temporal data locality



HDS Classification

3) HDS data layout

Latency/Block Size per one transfer



Hardware Note: SSD disks

- SSD = solid state drive
- In principle it is a flash disk emulating SATA interface
- No mechanical parts
- Two basic versions (MLC = multi level cell and SLC = single level cell)
- Pros: latency in order of microseconds
- Cons: price (significantly higher than for ordinary disks of the same capacity)

HDS Classification

3) HDS data layout

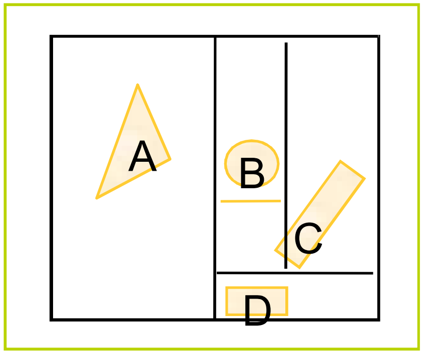
- Internal data structures (use only internal memory)
- External data structures (sometimes called „out of core“)
- Cache-aware data structures (knowing the block sizes and properties of hierarchy)
- Cache oblivious data structures (no cache parameters available, but the caching is assumed)

Node Types in HDS

- An interior node represents a *"pivot"* – according to pivot the data entities are sorted
- Typical content is a subdivision plane or a set of planes plus references to child nodes
- The content of interior node is crucial for the performance of the searching problem

Spatial Subdivisions

- **Non-overlapping regions of child nodes**
- Space is organized by some (cutting) entities, typically by planes, constructed top-down
- Fully covering an original spatial region, point location always possible in some (empty or non-empty) leaf
- They are often called space partitionings



kd-tree

Spatial Subdivision Examples

- Kd-trees – axis aligned planes
- BSP-trees – arbitrary planes
- Octrees – three axis aligned planes in a node
  (Quadtrees – two axis aligned planes)
- Uniform grids (regular subdivision)
- Recursive grids

Object Hierarchies

- Possibly **overlapping** extents of child nodes
- Many different names - often called bounding volume hierarchies
- Possibly some spatial regions are not covered by an object hierarchy - point location is then impossible
- Construction methods
  - top-down (sorting)
  - bottom-up (clustering)
  - incrementally (by insertion)

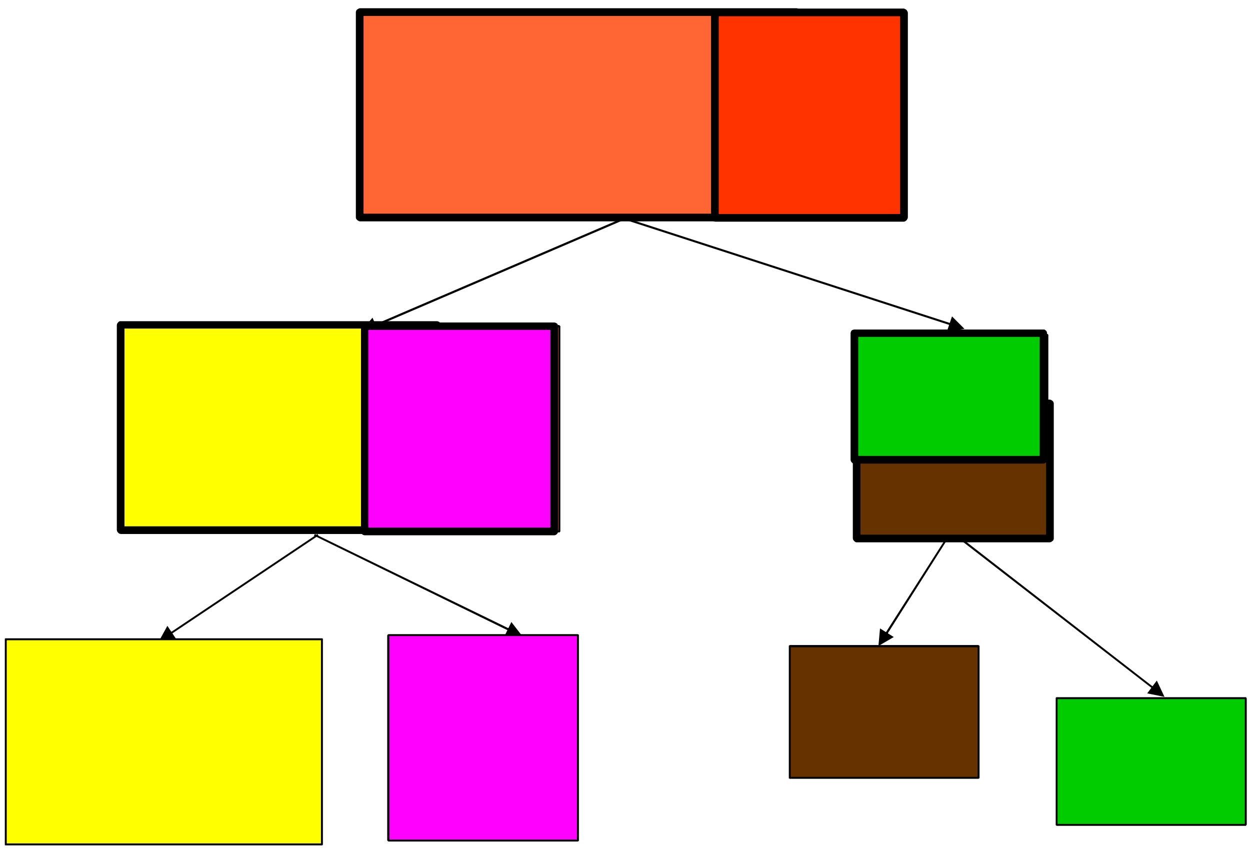
Names used for Object Hierarchies

- Bounding Volume Hierarchies (BVHs)
- R-trees and their many variants
- Box-trees
- Several others (special sort of bounding volumes... sphere trees etc.)

Bounding Volume Hierarchies

(The shape represented by interior nodes typically a box, but other shapes as spheres also possible)

Constructed Top-Down



Hybrid Data Structures

- Combining between various interior nodes
- E.g. possibly combining between spatial subdivisions and object hierarchies
- Sharing pros and cons of both types
- They can be tuned to compromise some properties, for example efficiency and memory requirements

Other HDS

- *Content of the node* – a single splitting plane, more splitting planes, a box, and some additional information.
- Arity of a node (also named as branching factor, fanout factor)
- A way of constructing a tree (height, weight balancing) + postprocessing
- Data only in leaves or also in interior nodes
- Augmenting data

Example of Other HDS

- Cell trees (polyhedral shapes for splitting)
- SKD-trees (two splitting planes at once)
- hB-trees (holey brick B-trees)
- LSD-tree (height balanced kd-tree)
- P-trees (polytope trees)
- BBD-trees (bounding box decomposition trees)
- And many others

(For details see book [Samet06])

HDS Transformation Approach

- *Input:* A spatial object in 2D or 3D domain, for example a box
- *Output:* A point in 4D or 6D domain
- More complicated mapping is possible, for example a sphere in 3D maps to a 4D point
- The transformation often changes the searching algorithm completely

Construction Algorithm (Top-Down)

Initial Phase: create a node with all elements and put it to the auxiliary structure AS (stack or priority queue).

Top-Down, Divide and Conquer:

(1) If AS is empty, then the algorithm stops.

(2) Take a node from an auxiliary data structure AS.

(2) Investigate a set of elements in the node and decide if to subdivide or not. If not to subdivide, then create a leaf and go to step (1).

(3) Decide how to split the set into two (N) subsets and create new nodes. Distribute the content to the nodes.

(4) Put the new nodes to AS. Go to step (1).

Search Algorithms using HDS

- Start from the root node
- Typically down traversal phase (location phase) + some other phase
- During visiting an interior node use either a stack (LIFO) or priority queue to record the nodes that are not visited now (but they are to be visited in future)
- Compute incidence (such as ray-object intersection) when visiting a leaf

Note: auxiliary structure implements another sorting phase during searching

Search Algorithms using HDS

- *Range queries* – given a range X, find all the incidences of X with data
- *Nearest neighbour* – find a nearest neighbor
- *k-nearest neighbour*
- *Intersection search* – given a point Q, find all the objects that contain Q
- *Ranking* – given a query object Q, report on all the objects in order of distance from Q
- *Reverse nearest neighbours* – given a point *Q,* find all the points to which Q is nearest neighbour

Search Performance Model

- Result = the cost of computation ... C
- Performance is proportional to the quality of the data structures for given problem
- The two uses of performance model
  - *a posteriori:* documenting and testing performance
  - *a priori:* constructing data structures with higher expected performance

Search Performance Model

Typical cost model:

$$C = C_T + C_L + C_R$$

$$C = C_{TS} * N_{TS} + C_{LO} * N_{LO} + C_{ACCESS} * N_{ACCESS}$$

- C_T … cost of traversing the nodes of HDS
- C_L … cost of incidence operation in leaves
- C_R … cost of accessing the data from internal or external memory

Performance Model

- C_T … cost of traversing the nodes of HDS
  - N_{TS} … number of traversal steps per query
  - C_{TS} … average cost of a single traversal step
- C_L … cost of incidence operation in leaves
  - N_{LO} … number of incidence operation per query
  - C_{LO} … average cost of incidence operation
- C_R … cost of accessing the data from internal or external memory
  - N_{ACCESS} … number of read operations from internal/external memory per query
  - C_{ACCESS} … average cost of read operation

HDS for Dynamic Data - Introduction

- Two major options:
  - Rebuild HDS after the data changes from scratch
  - Update only necessary part of HDS
    - ➔ Insertion method
    - ➔ Postorder processing
- Design considerations:
  - How much data are changed (M from N entities)
  - How efficient would be the updated data structures now and in the longer run?
  - How much time is required in both methods?

Rebuild from Scratch

- Construction time is typically O(N log N)
- The constants behind big-O notation are important in practice !
- Suitable if most objects are moving ($M \approx N$)
- Quality of hierarchy is high !
- Hint: (Top-down HDS) Number of exchange operations can be decreased significantly if we keep the order given by the previous hierarchy for incremental data changes.

HDS updates

- Given changes, only update data structures to reflect these changes
- It is assumed that the performance of searching remains acceptable after update, but no guarantees
- Updates requires additional bookkeeping data to monitor the cost/quality of a HDS node and the subtree associated with the node
- Techniques known for 1D trees (rotation, balancing) are often not applicable
- It is usually required to update larger amount of data at once (*bulk updating*)

HDS updates

- Insertion method – delete and reinsert the data in the tree (also deferred insertion)
  - Suitable if the number of changed objects is small
  - Each insertion/deletion requires O(log N)
  - Necessary delete and update some interior nodes
- Postorder processing (only for object hierarchies)
  - Suitable if number of changed objects is high
  - First update all leaves (data itself)
  - Traverse the whole tree in O(N) and reconstruct interior nodes of object hierarchy knowing both children
  - Possible structrural changes in the tree and further updates

Thank you for your attention!