The R*-tree: An Efficient and Robust Access Method for Points and Rectangles

Norbert Beckmann, Hans-Peter begel
Ralf Schneider, Bernhard Seeger

Presented by Snehal Thakkar

Contents

Terminology

- Bounding box
  - a.k.a. Minimum Bounding Rectangle MBR
  - smallest axis-parallel rectangle enclosing the SDT value
- Directory rectangle
  - geometrically the MBR of the underlying rectangles
- Margin
  - the sum of the lengths of the edges of a rectangle
- PAM (Point Access Method)
  - A data structure and associated algorithms primarily to search for points defined in multidimensional space.
- SAM (Spatial Access Method)
  - A data structure to search for lines, polygons, etc.

Abstract

- R-tree
  - based on the heuristic optimization of the area of the enclosing rectangle in each inner node
  - heuristic: an algorithm which usually, but not always, works or which gives nearly the right answer
- R*-tree
  - incorporates a combined optimization of area, margin and overlap of each enclosing rectangle in the directory
  - outperform existing R-tree variants
  - efficiently support point and spatial data at the same time
  - implementation cost is only slightly higher than other R-trees

Introduction

- Spatial Access Methods (SAMs)
  - approximate a complex spatial object by axis-parallel MBR
  - a complex object is represented by a limited number of bytes
  - Although a lot of information is lost, MBR of spatial objects already preserve the most essential geometric properties
    - the location of the object
    - the extension of the object in each axis
  - most popular SAM for storing rectangles: R-tree

R-trees

- Based on the PAM B+-tree using overlapping regions
- Optimization criterion: minimize the area of each enclosing rectangle in the inner nodes
- M: max. # of entries in one node
- m: min. # of entries in one node ($2 \leq m \leq M/2$)
- Satisfy the followings:
  - the root has at least two children unless it is a leaf
  - every non-leaf node has between m and M children unless it is the root
  - every leaf node contains between m and M entries unless it is the root
  - All leaves appear on the same level

R-trees

- An R-tree (R*-tree) is completely dynamic
  - insertions and deletions can be intermixed with queries
  - no periodic global reorganization is required
- The structure must allow overlapping directory rectangles
  - can not guarantee that only one search path is required for an exact match query
- This paper shows that the overlapping-regions-technique does not imply bad average retrieval performance

Main Problem in R-trees

- Known parameters of good retrieval performance affect each other in a very complex way
  - impossible to optimize one of them without influencing others
- Data rectangles have different sizes/shapes and the directory rectangles grow and shrink dynamically
  - the success of methods which will optimize one parameter is very uncertain
- A heuristic approach is applied
  - based on many different experiments carried out in a systematic framework

Optimization Criteria

(O1) Minimize area covered by a directory rectangle

- minimize dead space
- improve performance since decisions which paths have to be traversed can be taken on higher levels

(O2) Minimize overlap between directory rectangles

- decrease number of paths to be traversed

(O3) Minimize margin of a directory rectangle

- directory rectangle will be shaped more quadratic
- quadratic objects can be packed easier
- smaller directory rectangles in the level above

(O4) Optimize storage utilization

- height of the tree will be kept low
- reduce the query cost

Optimization

(+) Minimize area

- less covering of data space, hence overlap may be reduced

(+) More quadratic

- better packing, hence easier to maintain high storage utilization

(–) Minimize area and overlap

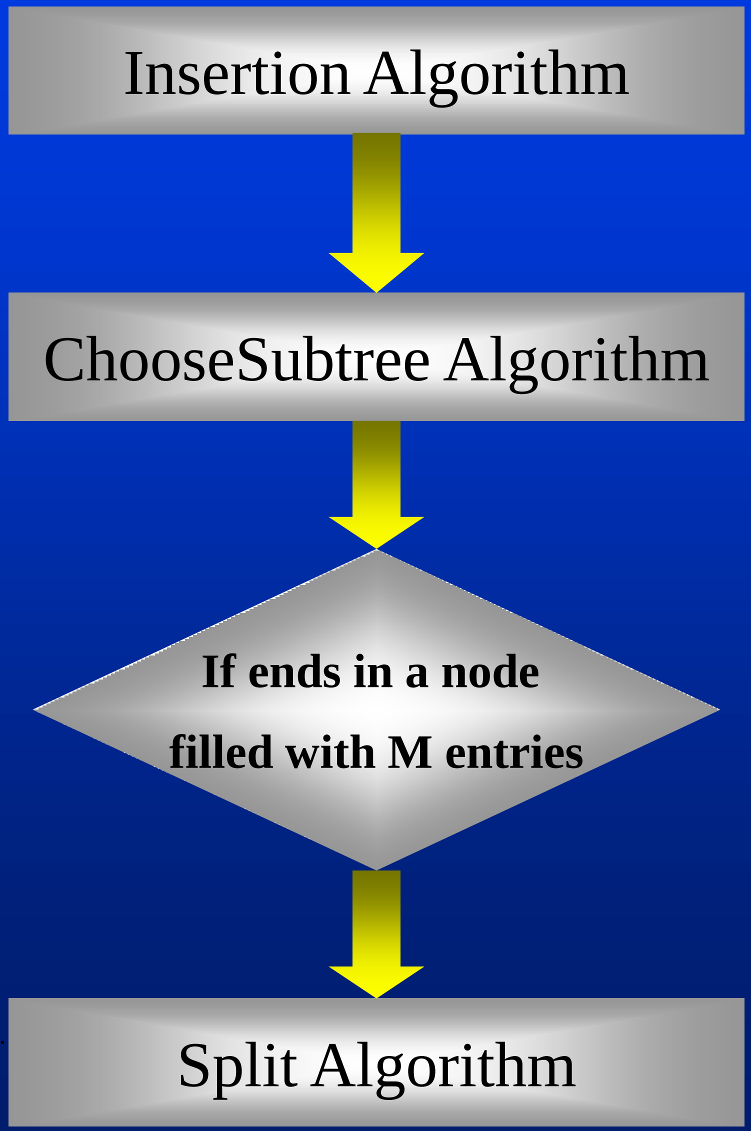
- require more freedom in # of rectangles stored in one node
- lower storage utilization
- “less quadratic” rectangles

(–) Minimize margin

- reduce storage utilization

R-tree Variants

Optimize retrieval performance during insertion



Descend from roof to leaf. Find on every level the most suitable subtree to accommodate the new entry

Distribute M+1 rectangles into two nodes in most appropriate manner

Original R-tree (by Guttman)

- Method of optimization
  - minimize area covered by a directory rectangle
  - may also reduce overlap
  - cpu cost will be relatively low

Original R-tree (by Guttman)

Algorithm ChooseSubtree

CS1 [Initialize] Set N to be the root node

CS2 [Leaf check]

If N is a leaf,

return N

else

[Choose subtree]

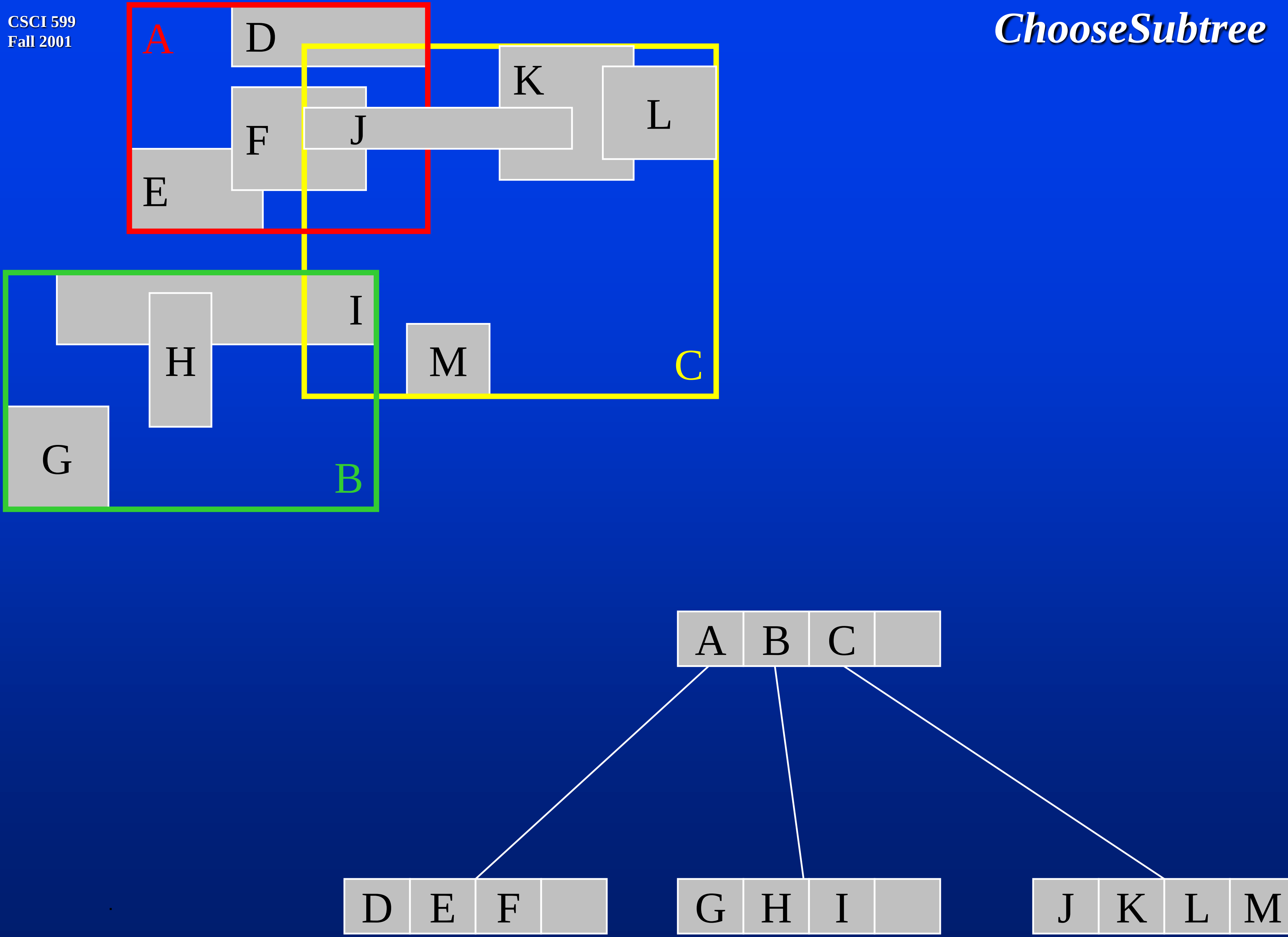
Choose the entry in N whose rectangle needs least area enlargement to include the new data. Resolve ties by choosing the entry with the rectangle of smallest area

end

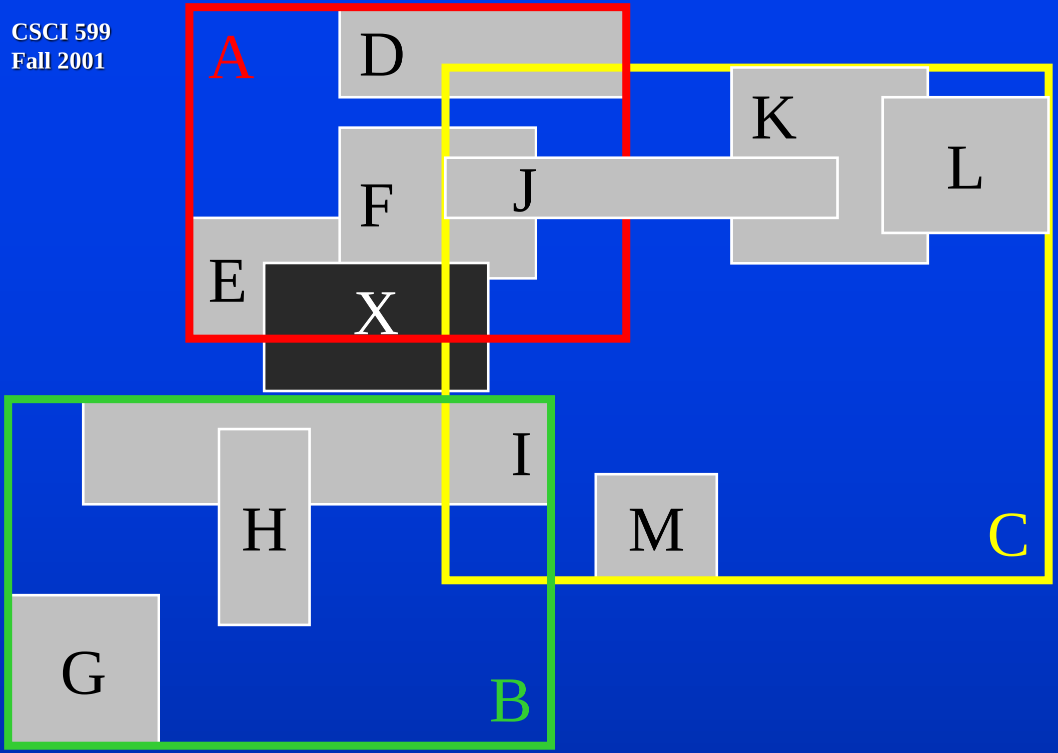
CS3 [Descend until a leaf is reached]

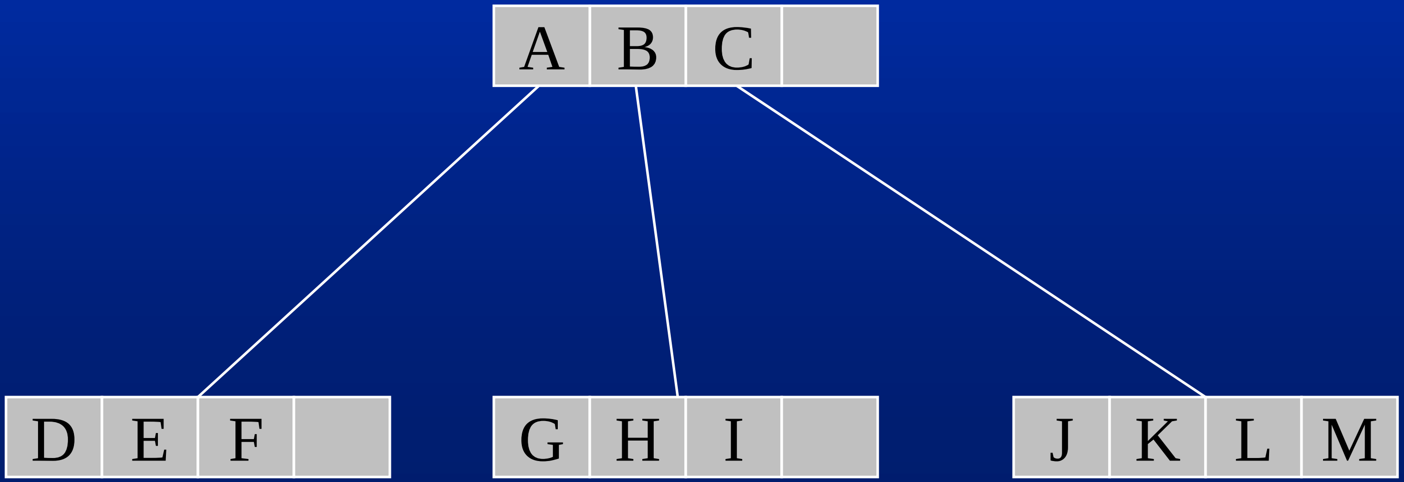
Set N to be the childnode pointed to by the childpointer of the chosen entry. Repeat from CS2

ChooseSubtree

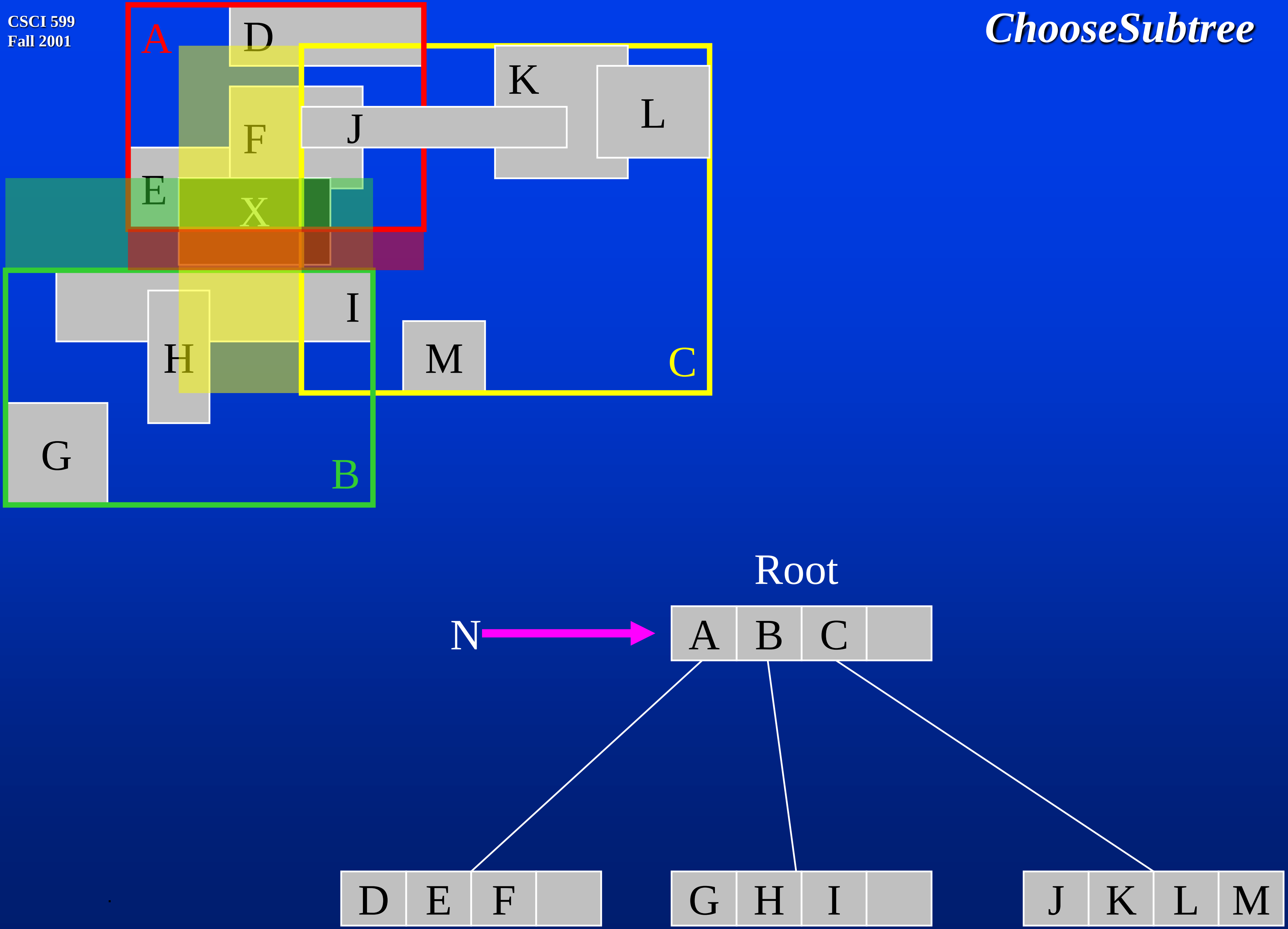


ChooseSubtree

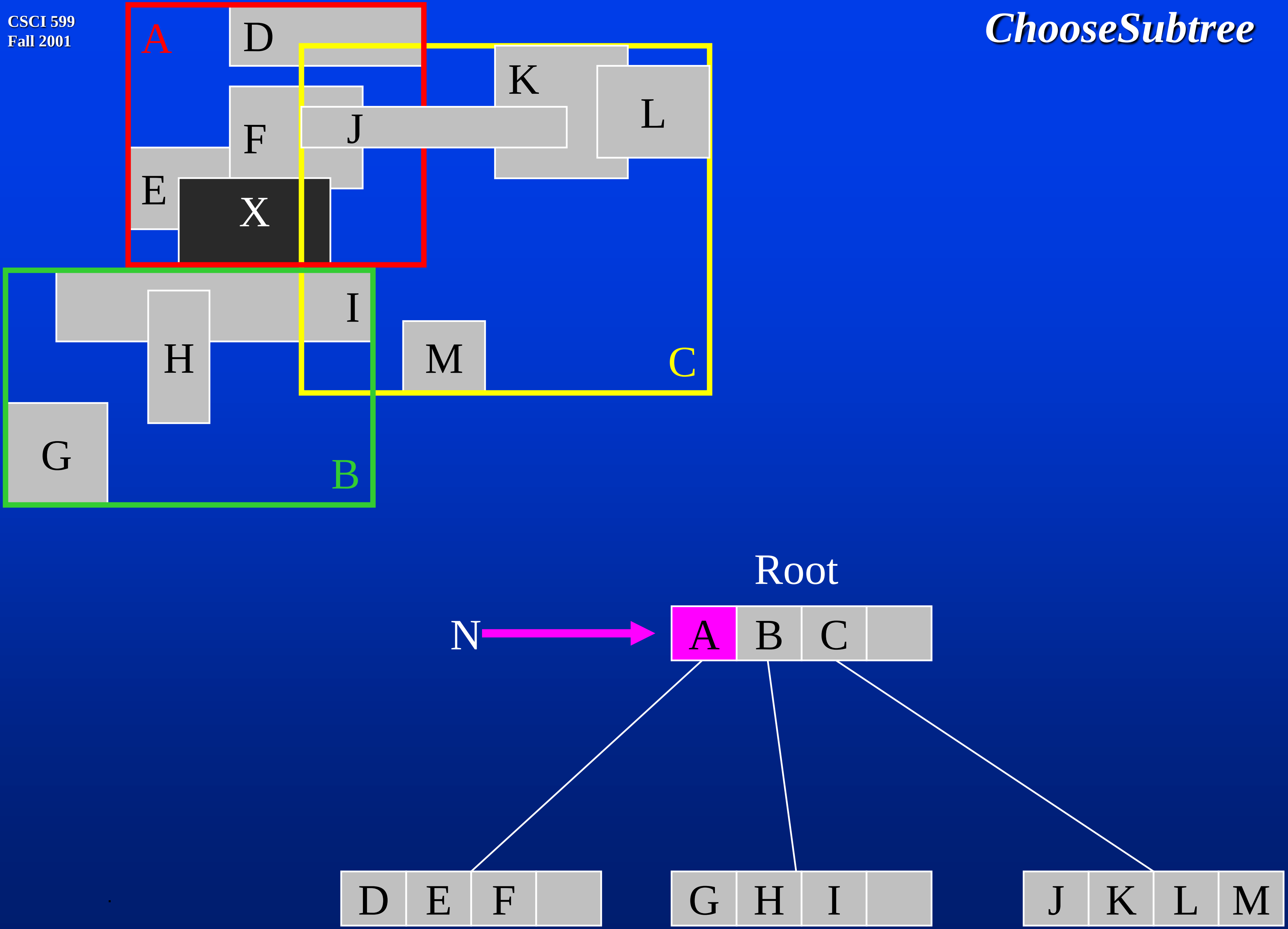




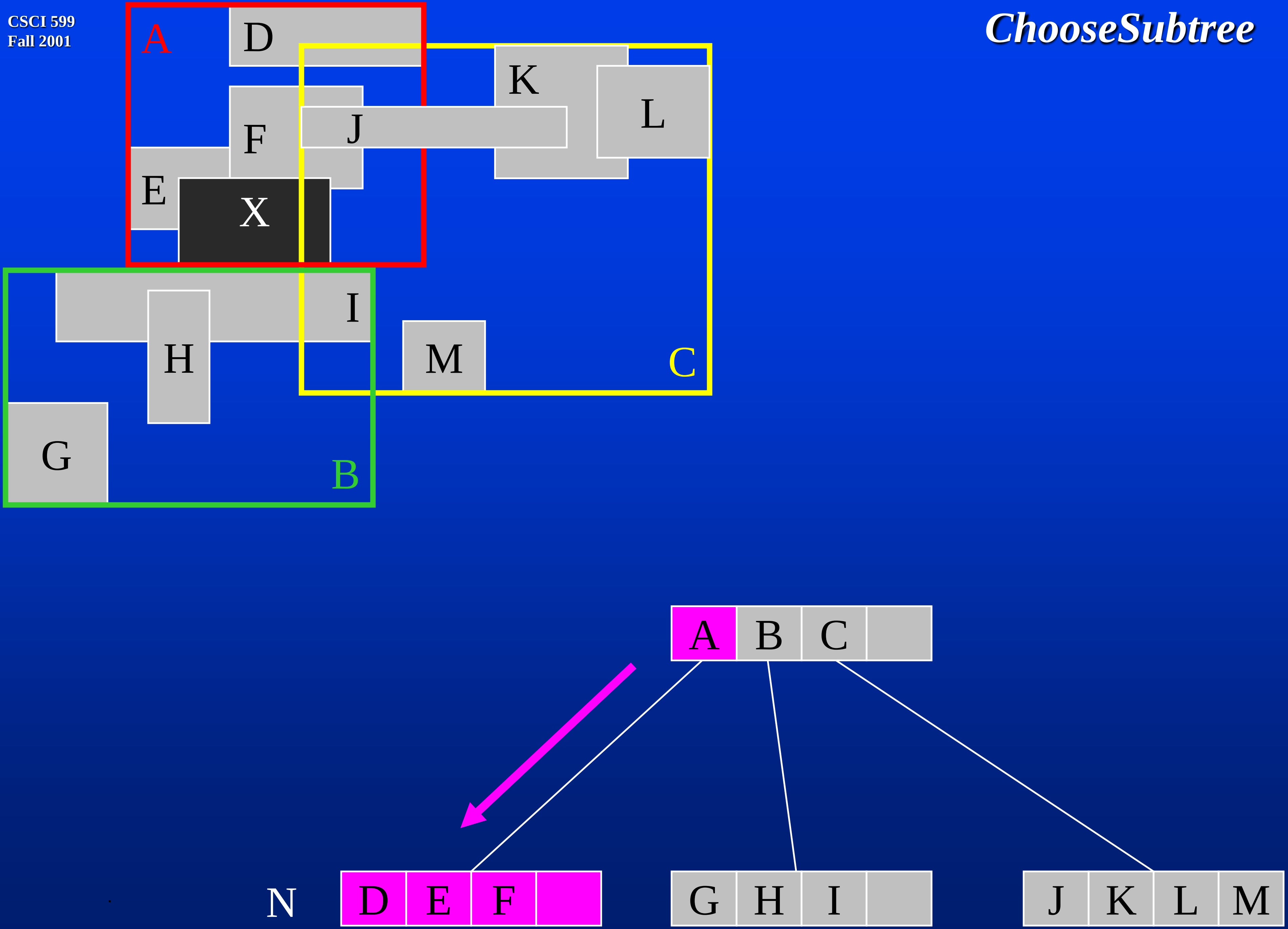
ChooseSubtree



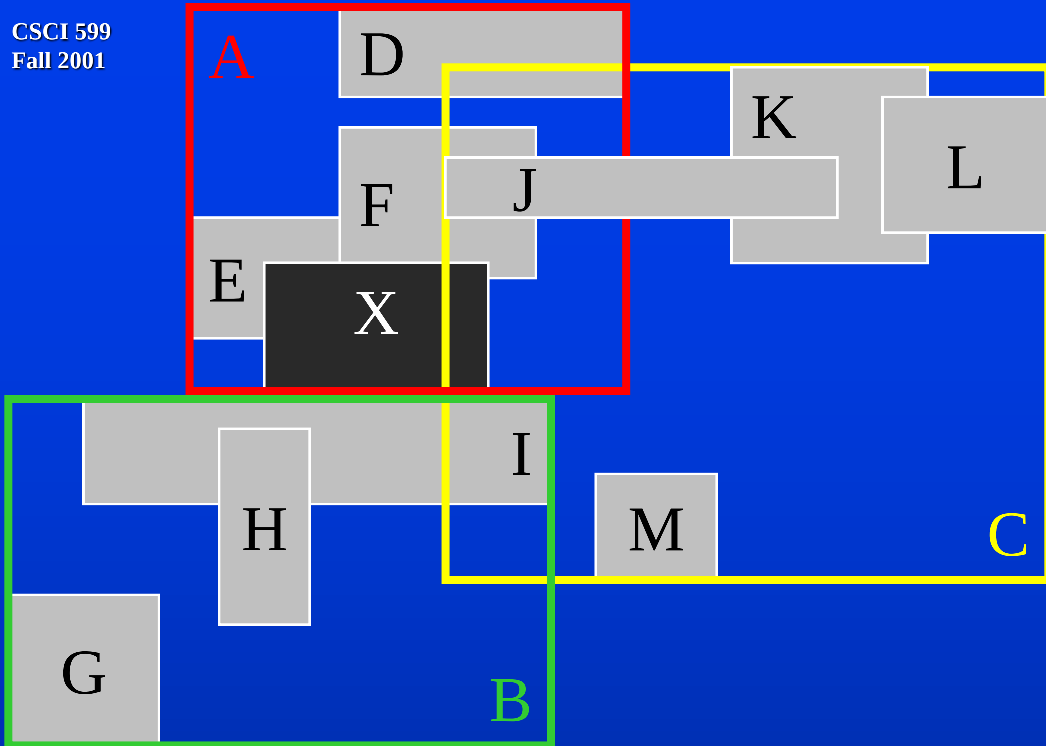
ChooseSubtree

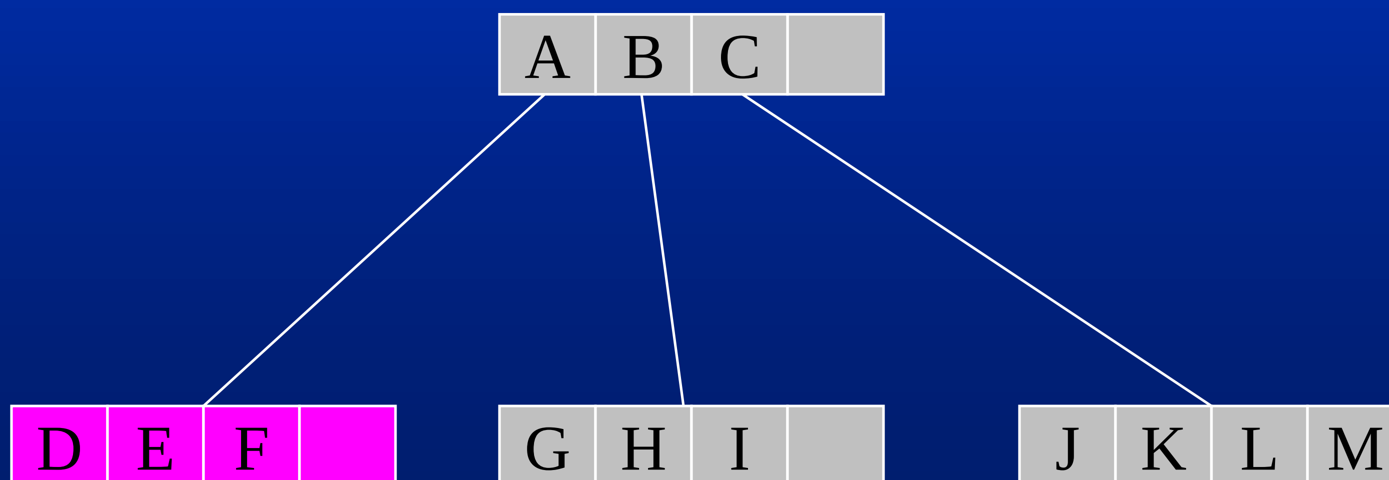


ChooseSubtree



ChooseSubtree

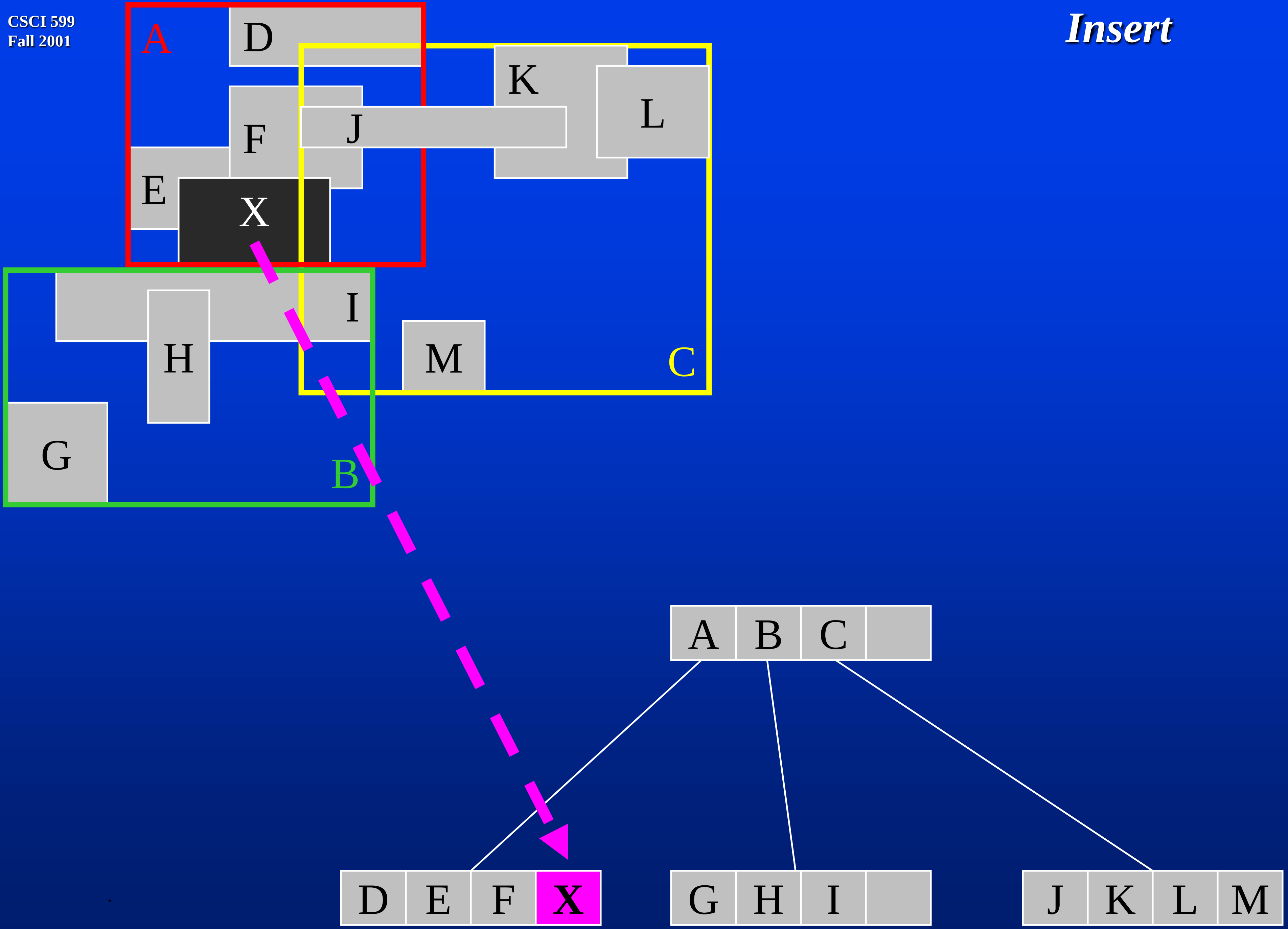




Since N = Leaf,

stop and return N

Insert



Split Algorithms

Three versions

- all are designed to minimize area covered by two rectangles resulting from split

❶ Exponential

- find the area with global minimum
- CPU cost too high

❷ Quadratic and ❸ Linear

- find approximation
- [Guttman] nearly same retrieval performance
- [this paper] quadratic performs much better than linear

Algorithm QuadraticSplit

[Divide a set of M+1 index entries into two groups]

QS1 [Pick first entry for each group]

Invoke PickSeeds to choose two entries, each be first entry of each group

QS2 [Check if done]

Repeat

DistributeEntry

until

all entries are distributed or one of the two groups has M–m+1 entries (so that the other group has m entries)

QS3 [Select entry to assign]

If entries remain, assign them to the other group so that it has the minimum number m required

Algorithm PickSeeds

[Choose two entries to be the first entries of the groups]

PS1 [Calculate inefficiency of grouping entries together]
For each pair of entries E1 and E2, compose a rectangle R including E1 rectangle and E2 rectangle

Calculate d = area(R) - area(E1 rectangle) - area(E2 rectangle)

PS2 [Choose the most wasteful pair]

Choose the pair with the largest d

[the seeds will tend to be small, if the rectangles are of very different size (and) or the overlap between them is high]

Algorithm DistributeEntry

[Assign the remaining entries by the criterion of minimum area]

DE1 Invoke PickNext to choose the next entry to be assigned

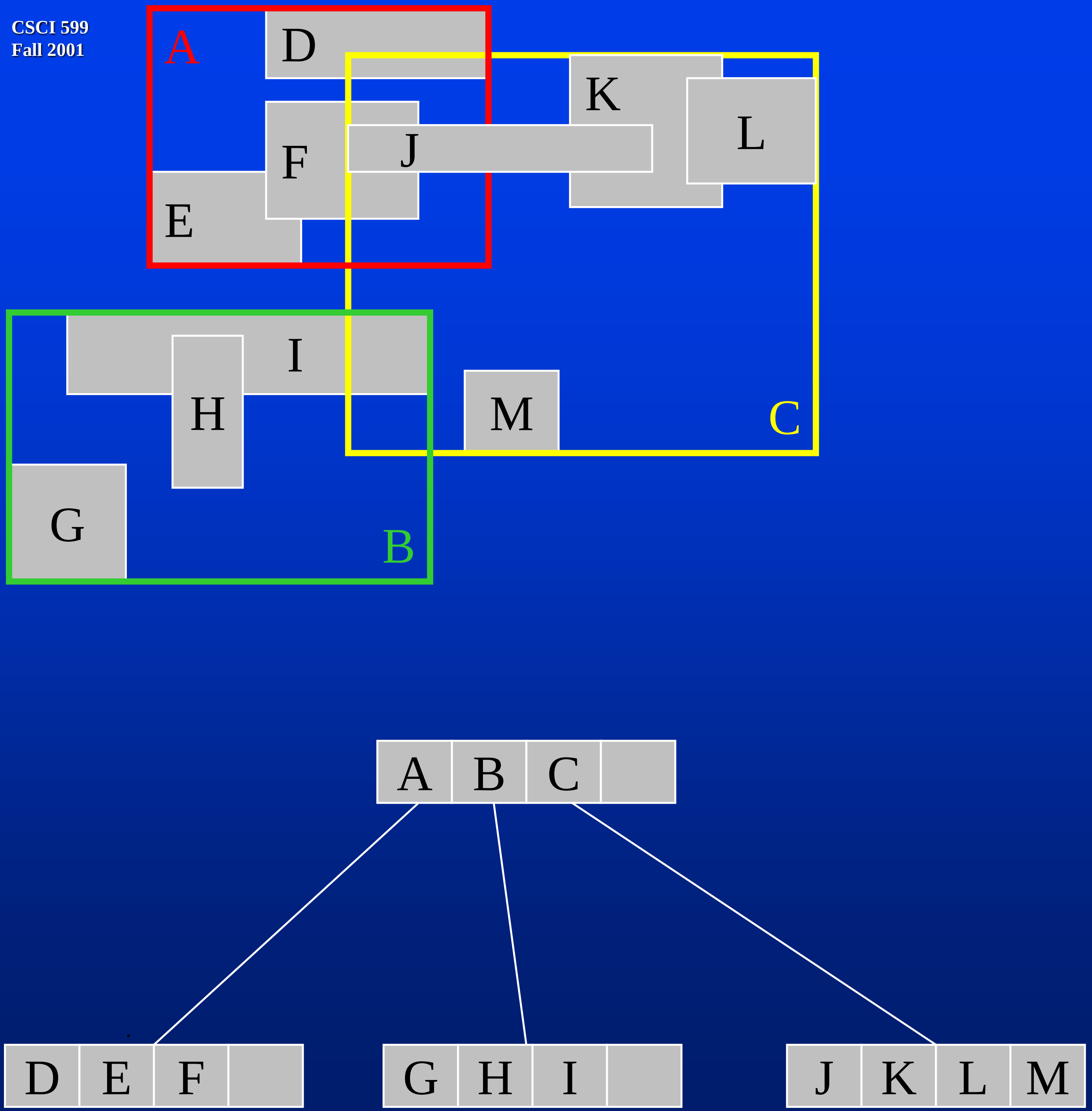
DE2 Add It to the group whose covering rectangle will have to be enlarged least to accommodate It. Resolve ties by adding the entry to the group with the smallest area, then to the one with the fewer entries, then to either

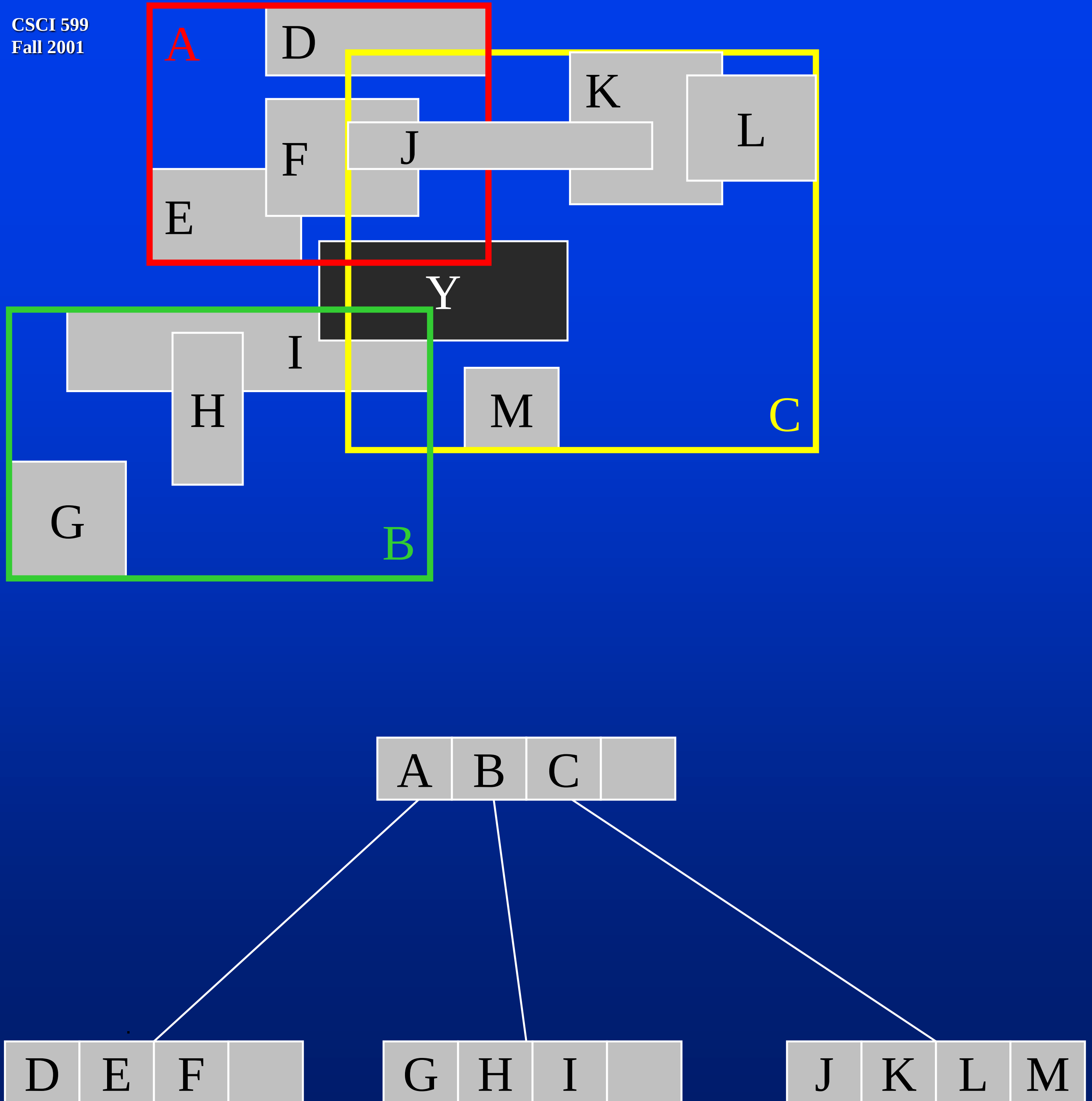
Algorithm PickNext

[chooses the entry with best area-goodness-value in every situation]

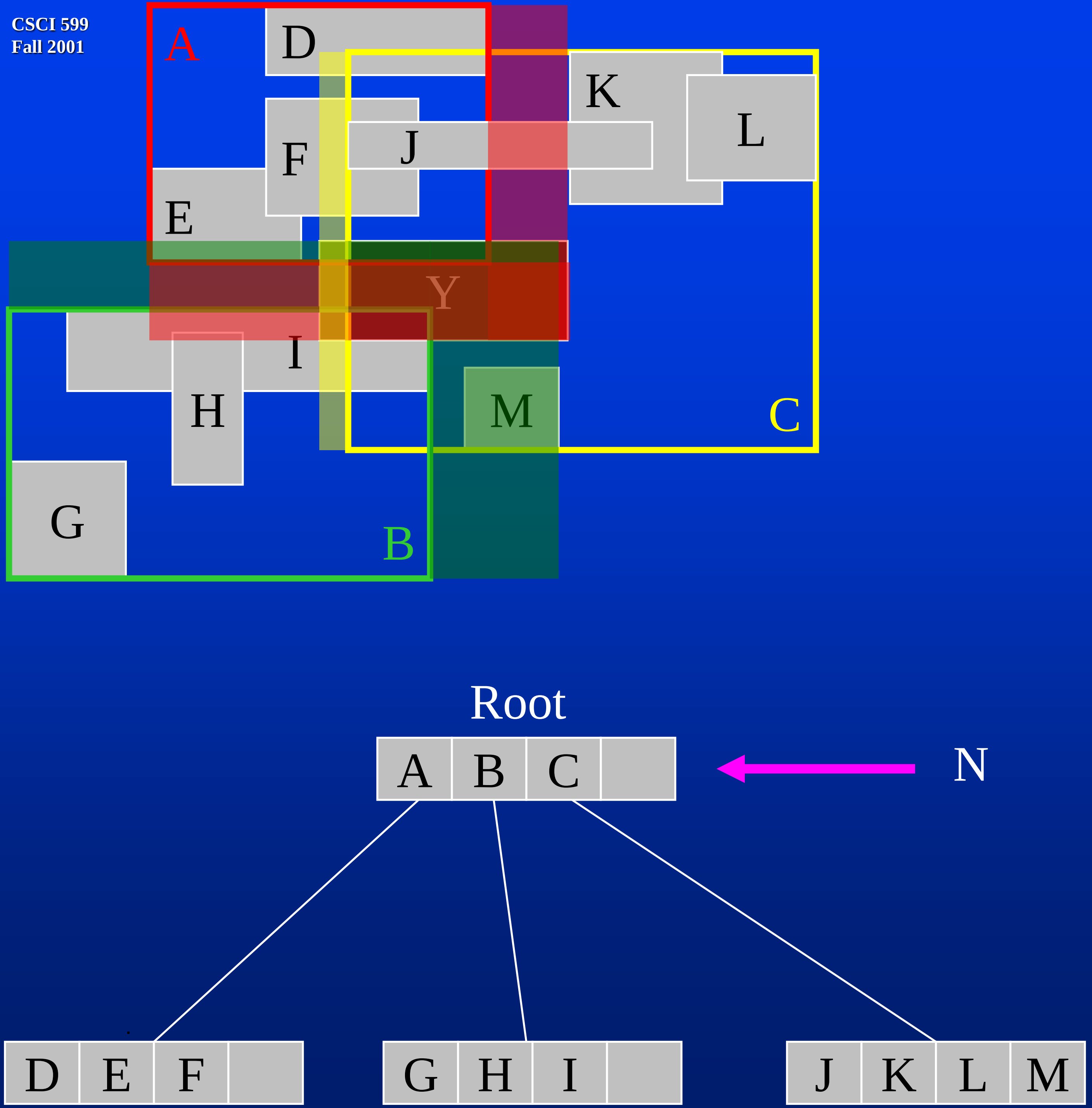
DE1 For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include E Rectangle. Calculate d_2 analogously for Group 2

DE2 Choose the entry with the maximum difference between d_1 and d_2

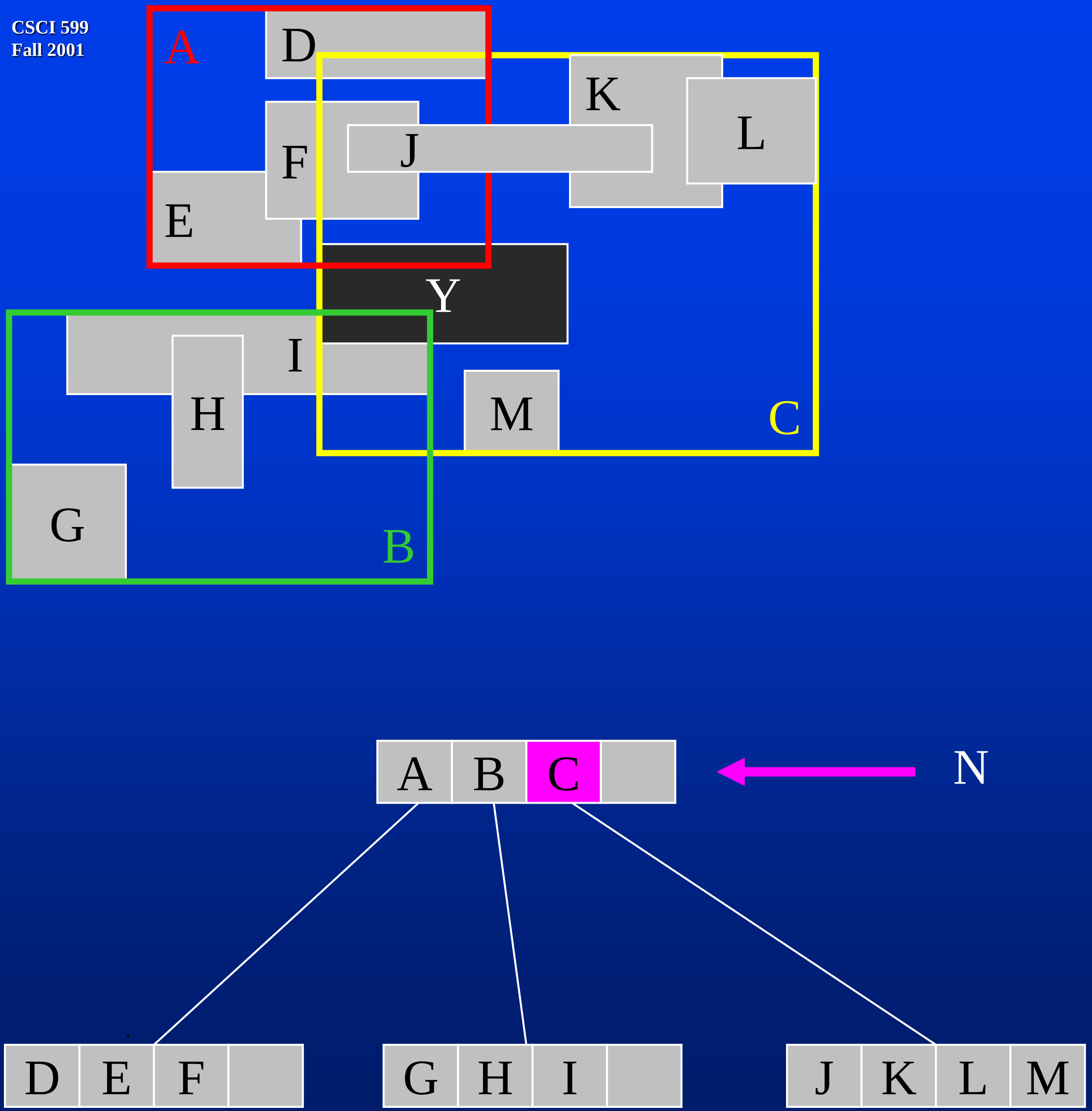

A
D
K
L
F
J
E
I
H
M
C
G
B
A
B
C
D
E
F
G
H
I
J
K
L
M


A
D
K
L
F
J
E
Y
I
H
M
C
G
B
A B C
D E F
G H I
J K L M

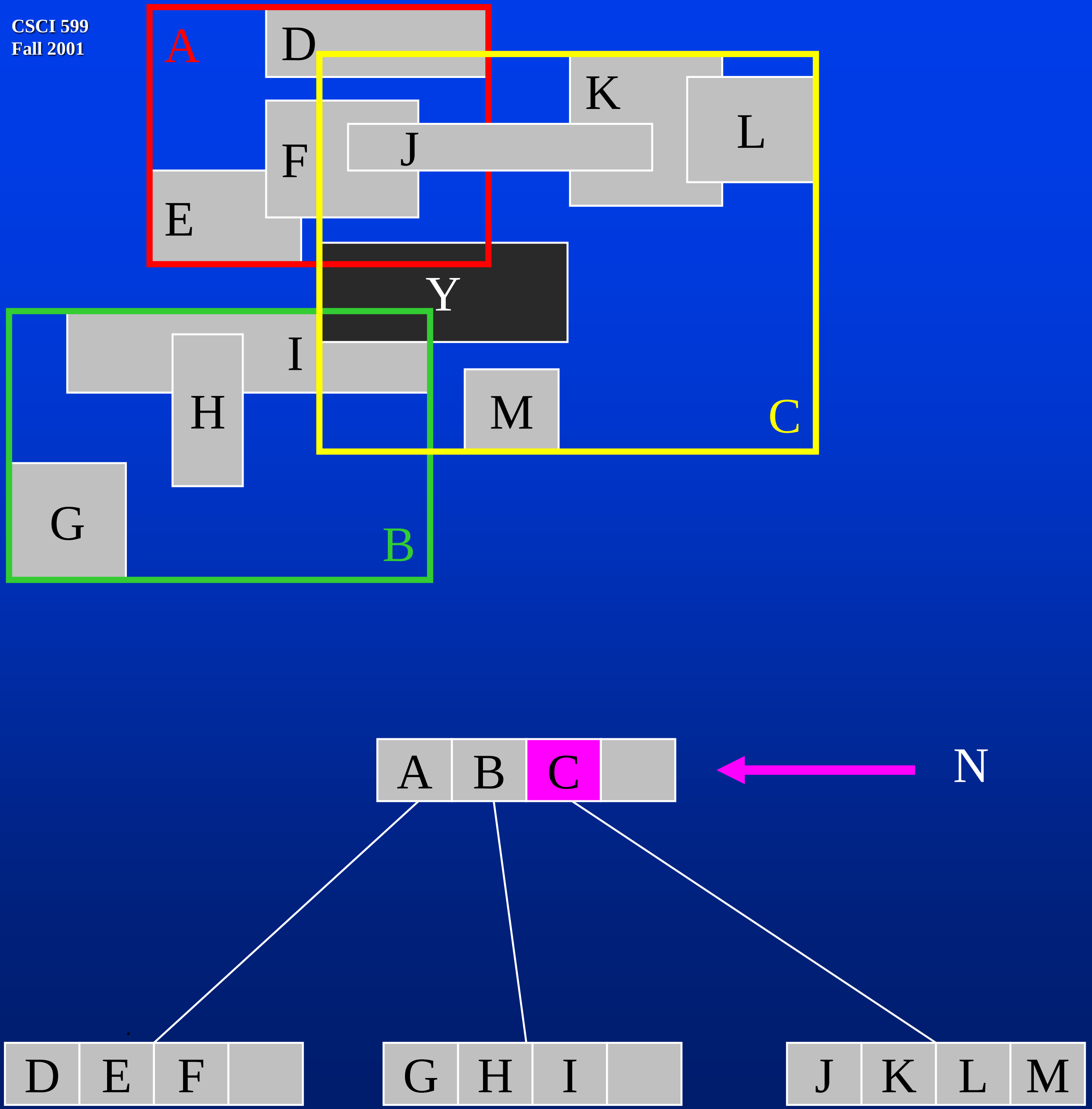
ChooseSubtree



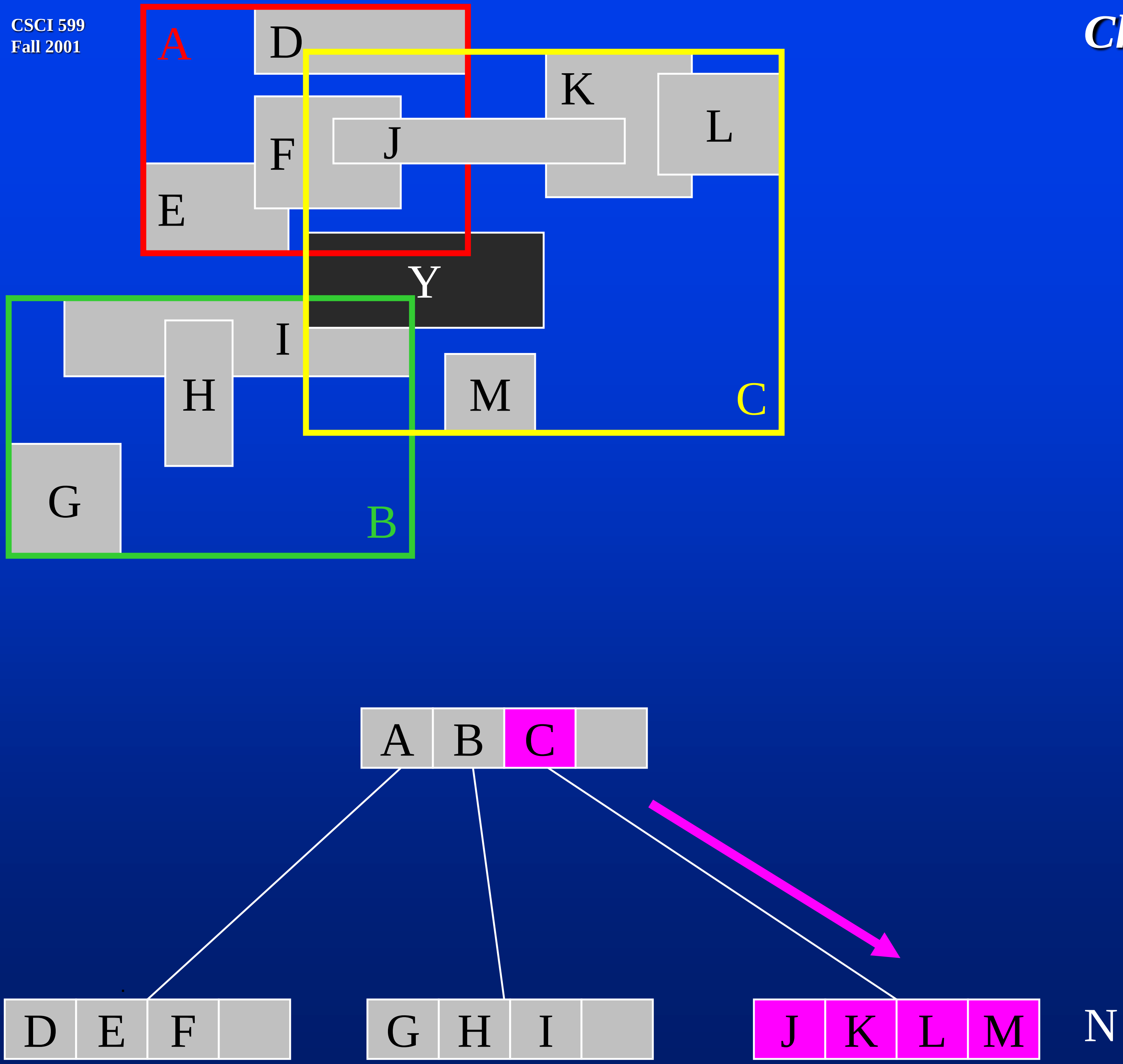
ChooseSubtree



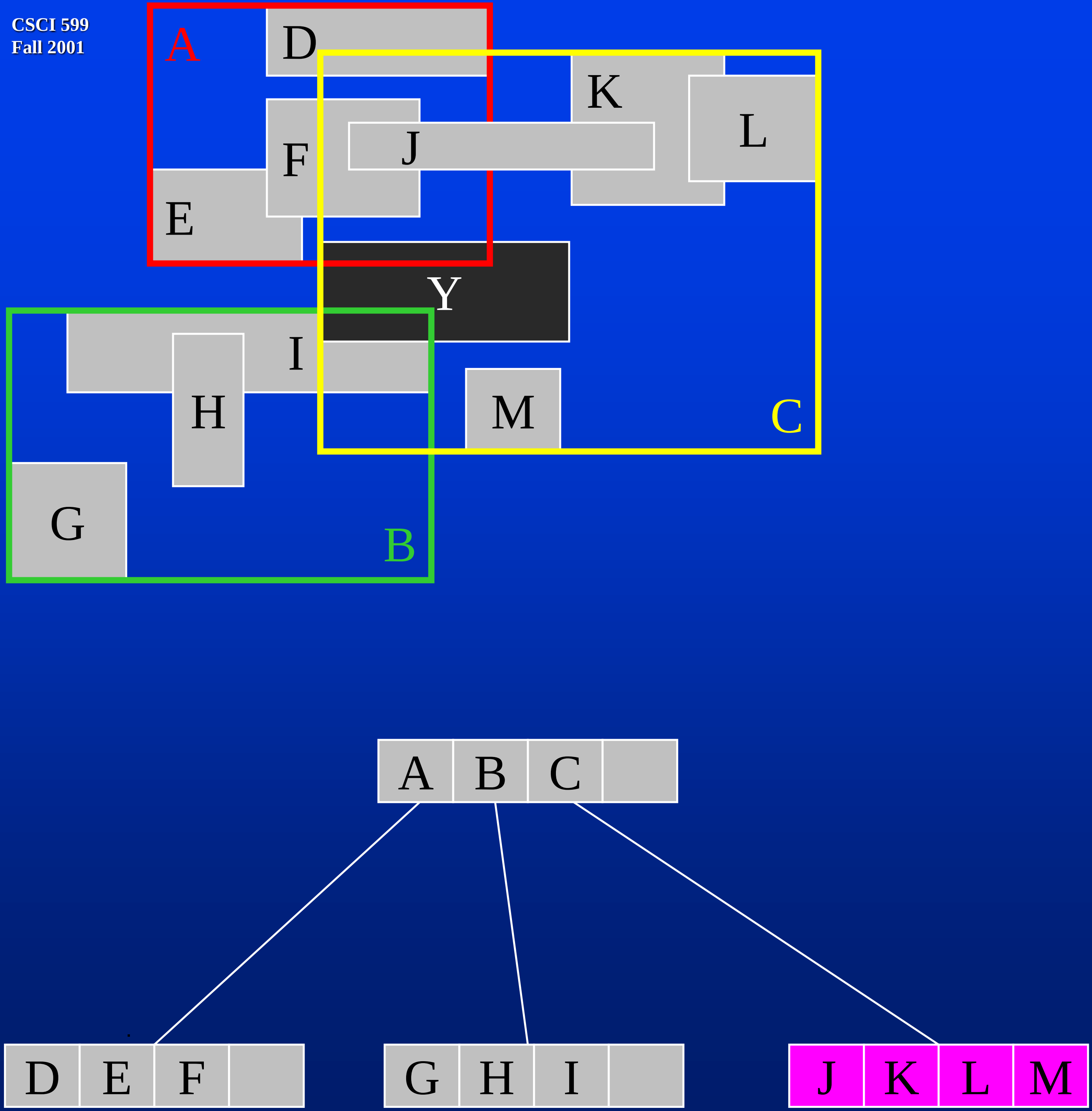
ChooseSubtree



ChooseSubtree



ChooseSubtree

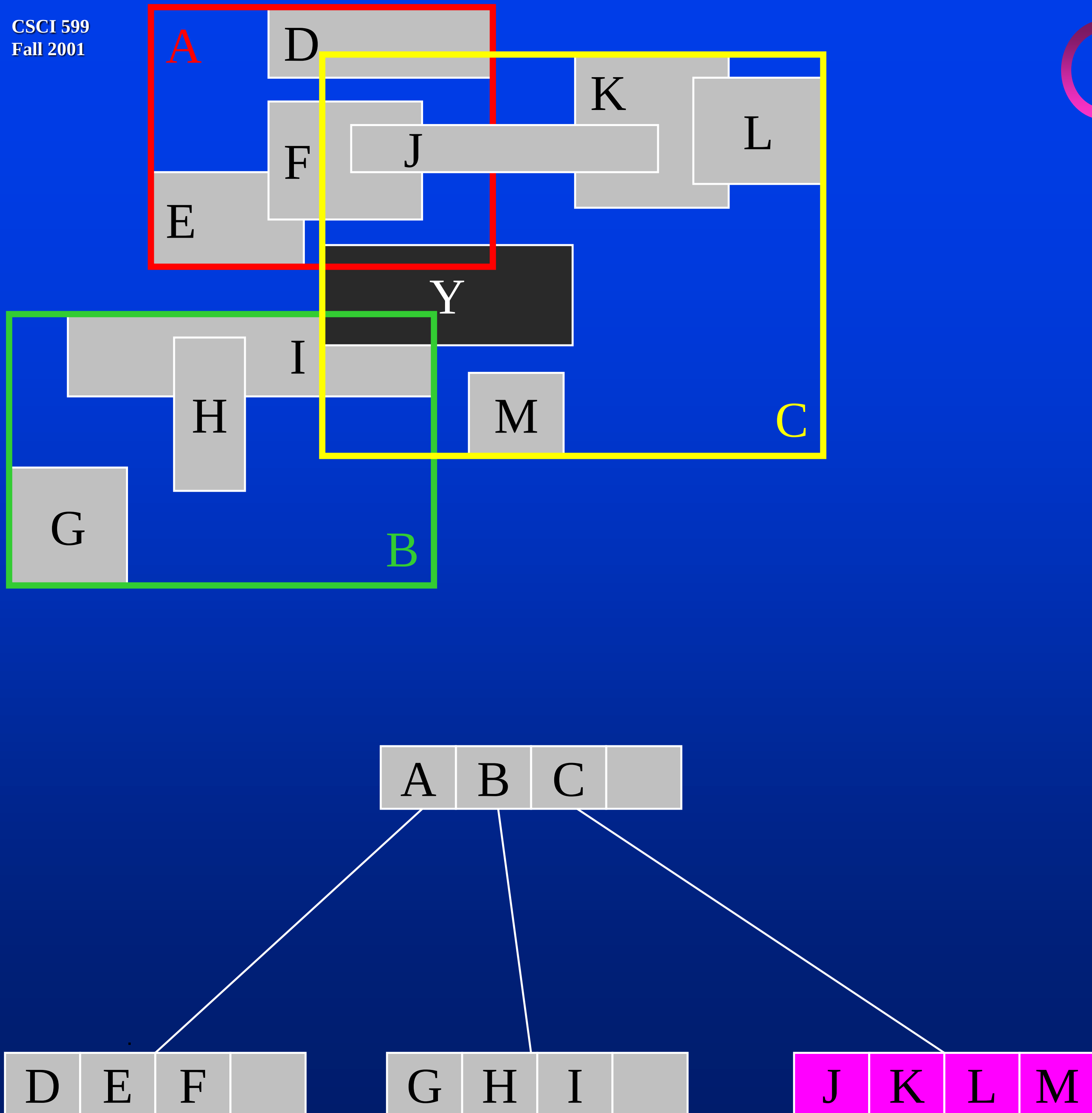


N = Leaf

Node is full

ChooseSubtree

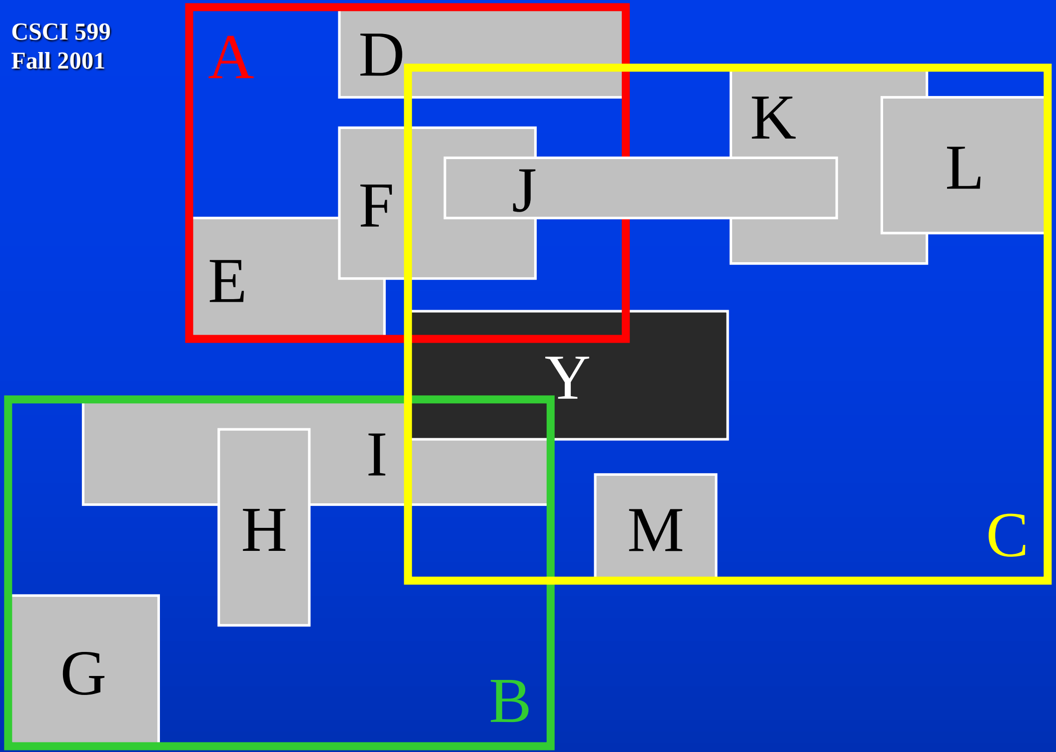
QuadraticSplit



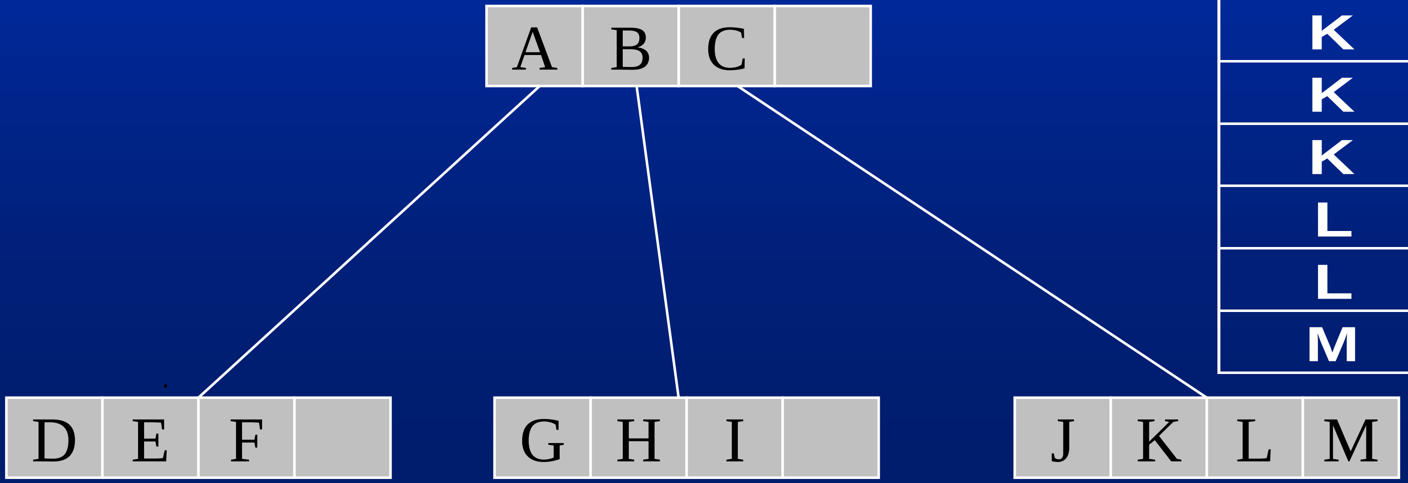
ChooseSubtree

QuadraticSplit

PickSeeds



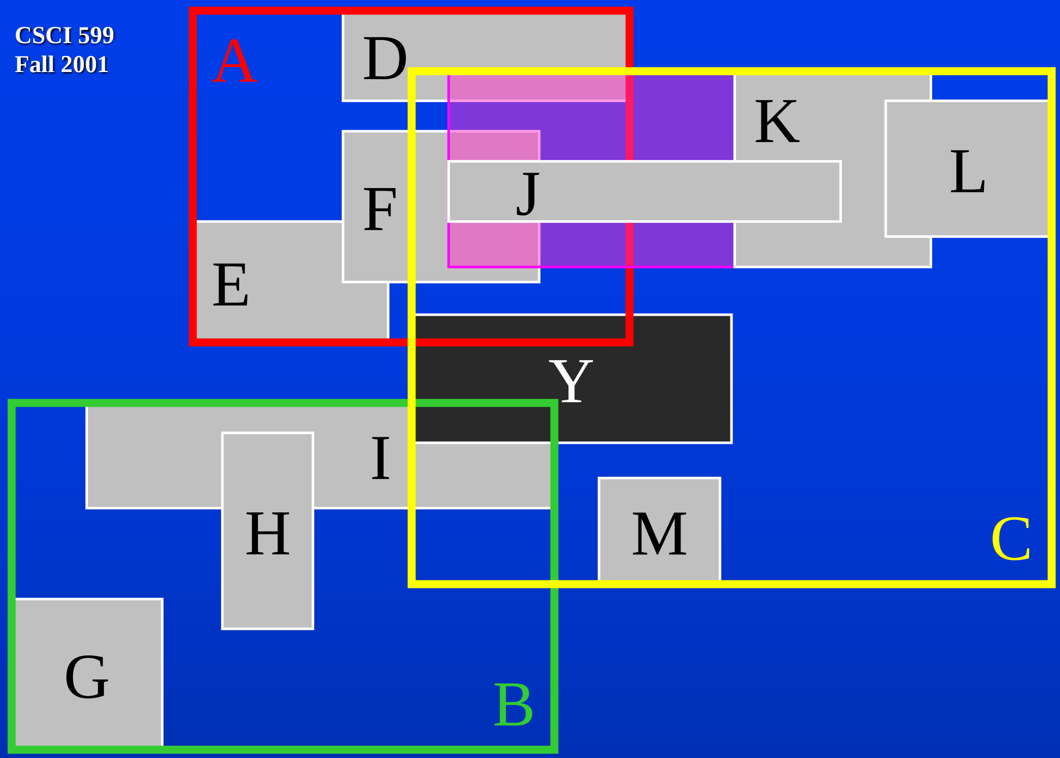
E1	E2	d
J	K	
J	L	
J	M	
J	Y	
K	L	
K	M	
K	Y	
L	M	
L	Y	
M	Y	



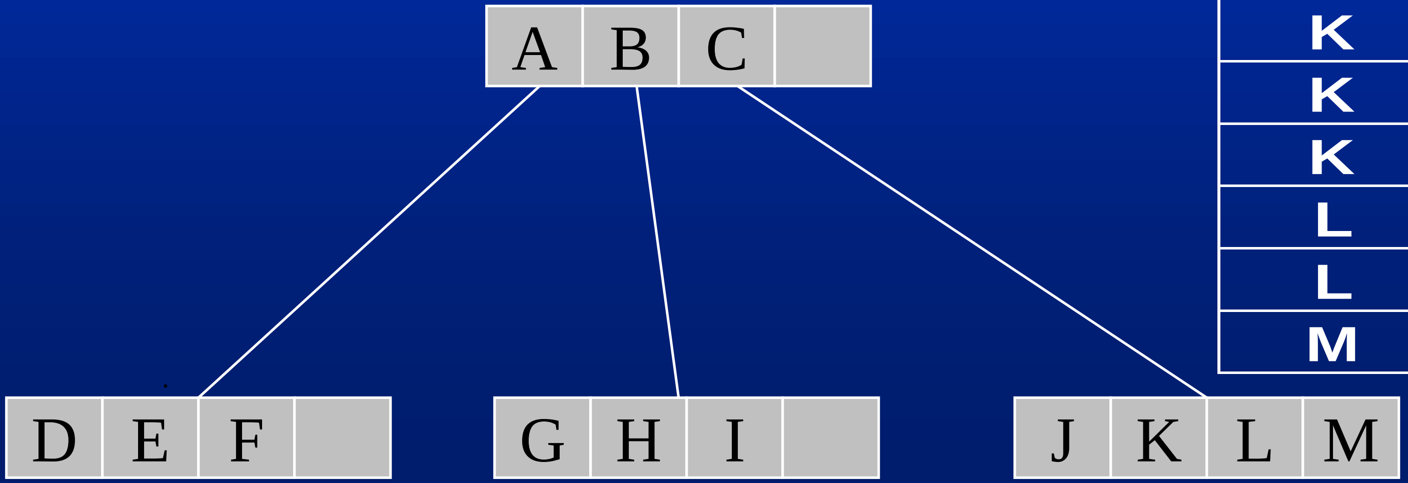
ChooseSubtree

QuadraticSplit

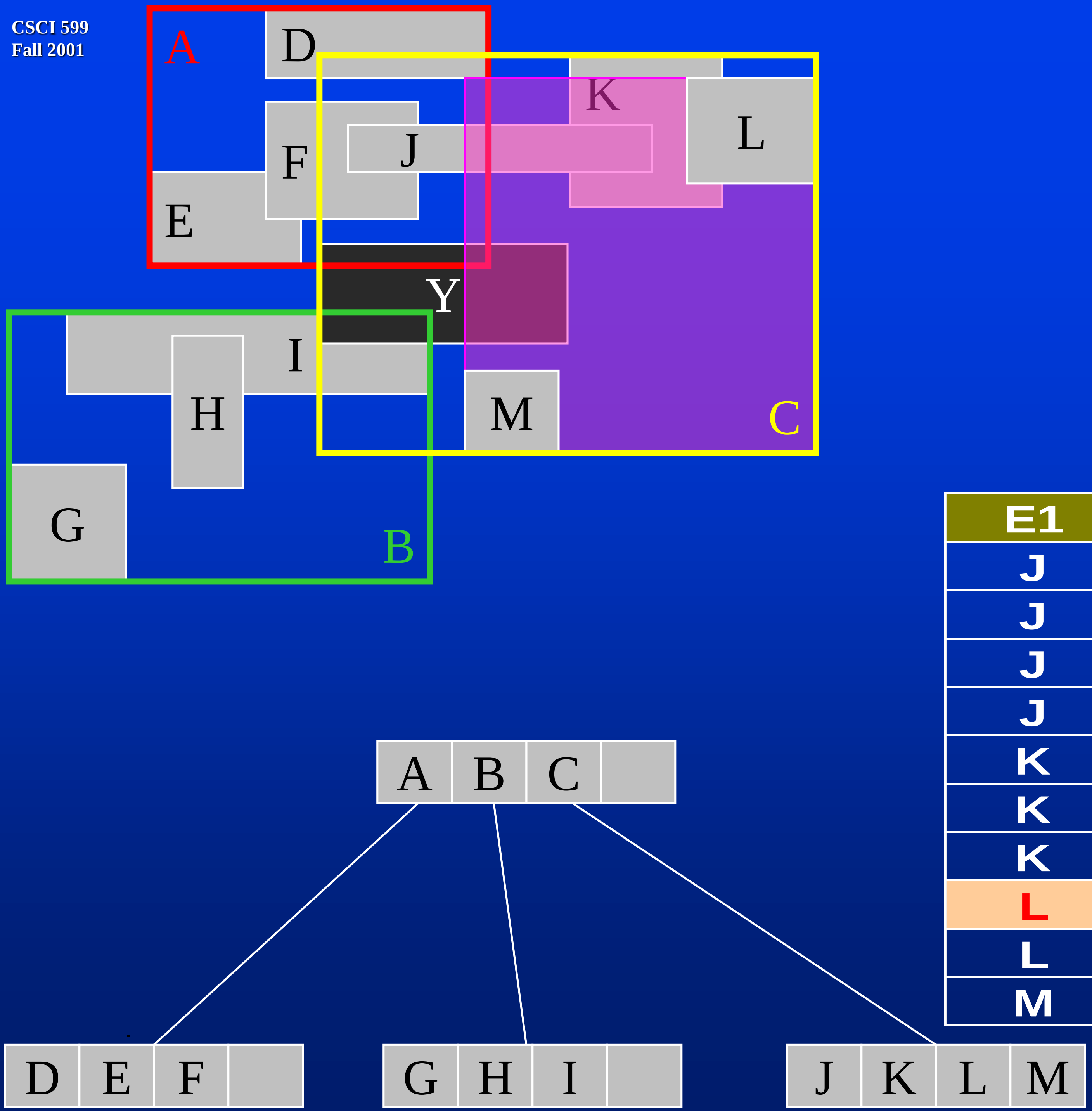
PickSeeds



E1	E2	d
J	K	1.0
J	L	
J	M	
J	Y	
K	L	
K	M	
K	Y	
L	M	
L	Y	
M	Y	

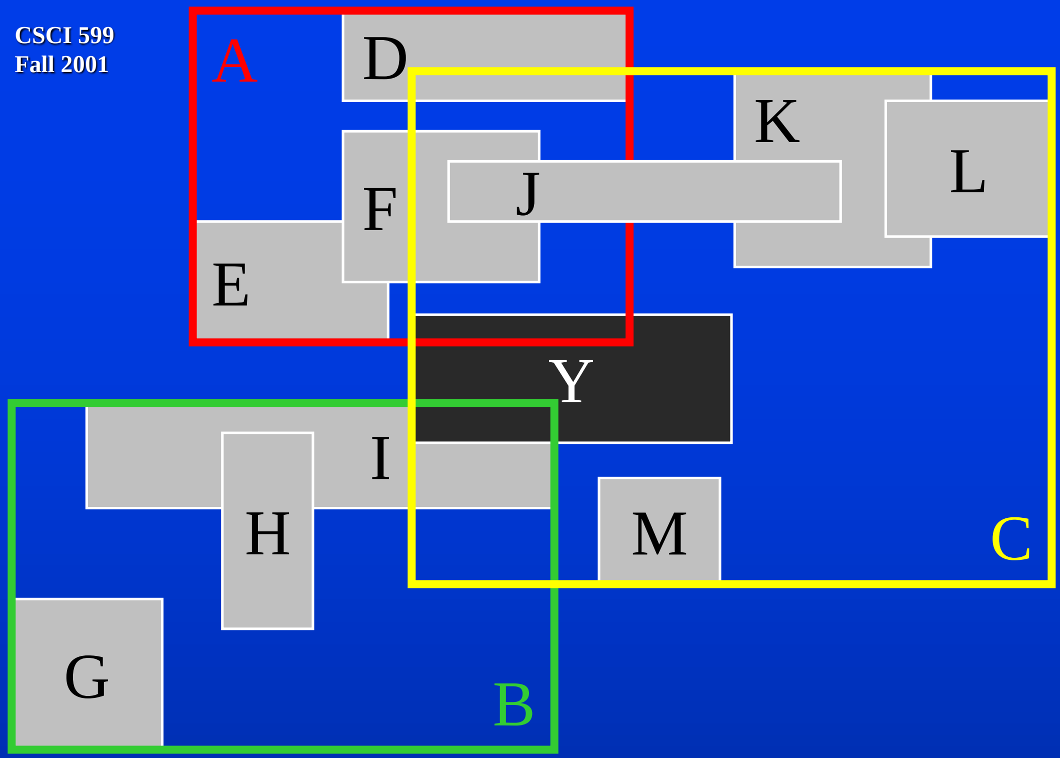


ChooseSubtree
QuadraticSplit
PickSeeds



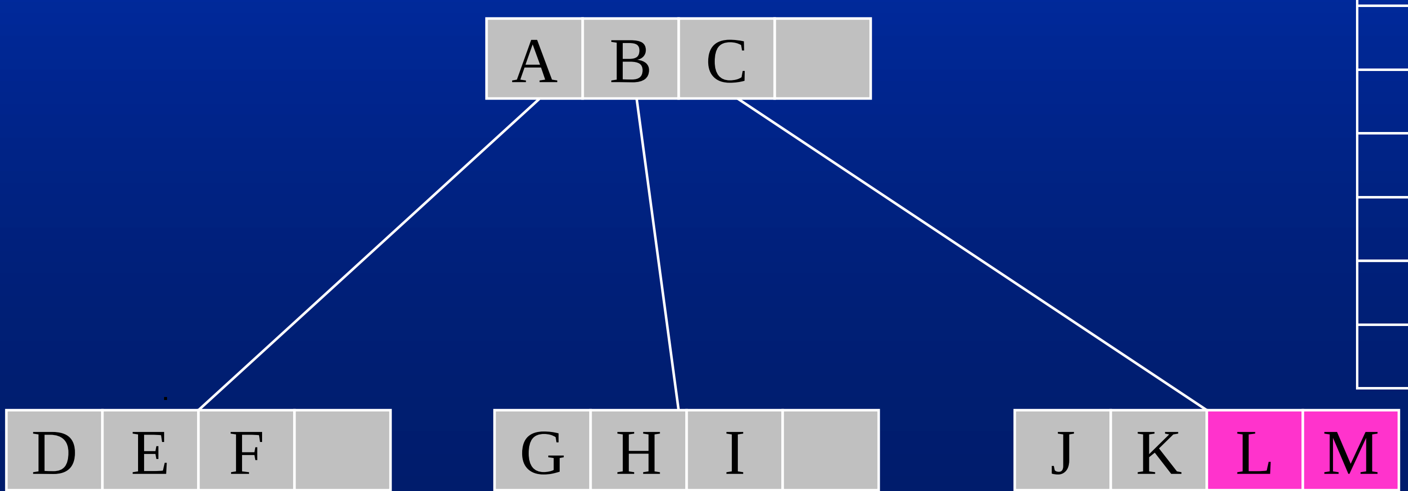
E1	E2	d
J	K	1.0
J	L	1.1
J	M	4.0
J	Y	1.7
K	L	0.1
K	M	3.6
K	Y	3.5
L	M	5.6
L	Y	4.7
M	Y	0.9

ChooseSubtree
QuadraticSplit
PickSeeds



G1	G2
L	M

E1	E2	d
J	K	1.0
J	L	1.1
J	M	4.0
J	Y	1.7
K	L	0.1
K	M	3.6
K	Y	3.5
L	Y	4.7
M	Y	0.9

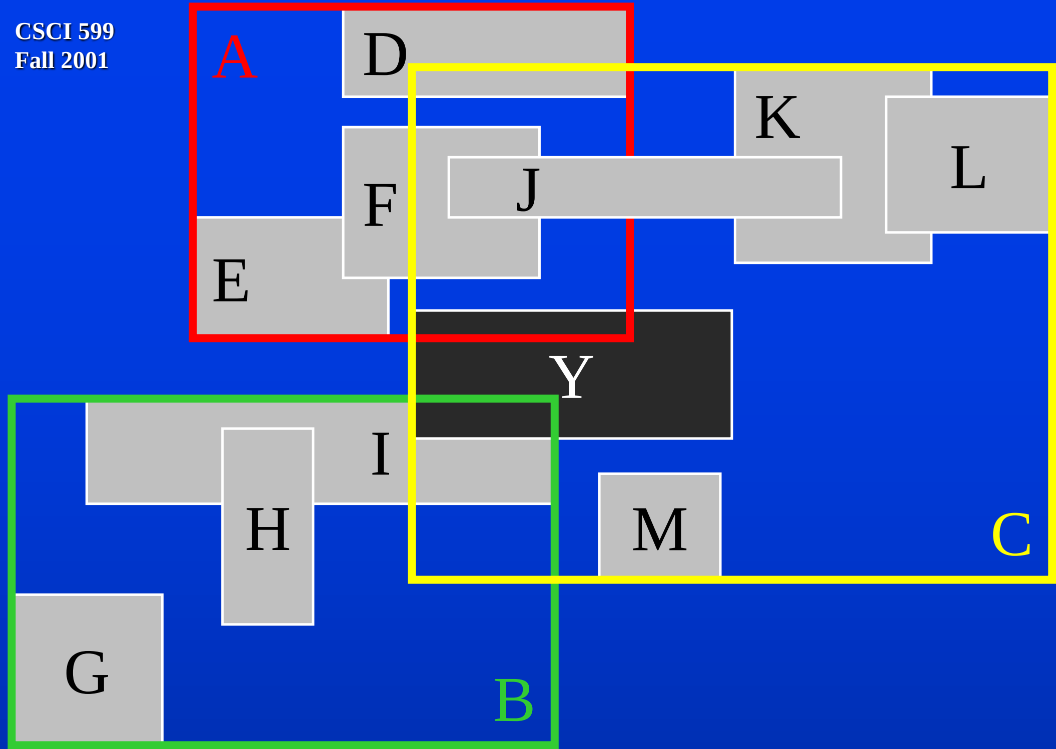


ChooseSubtree

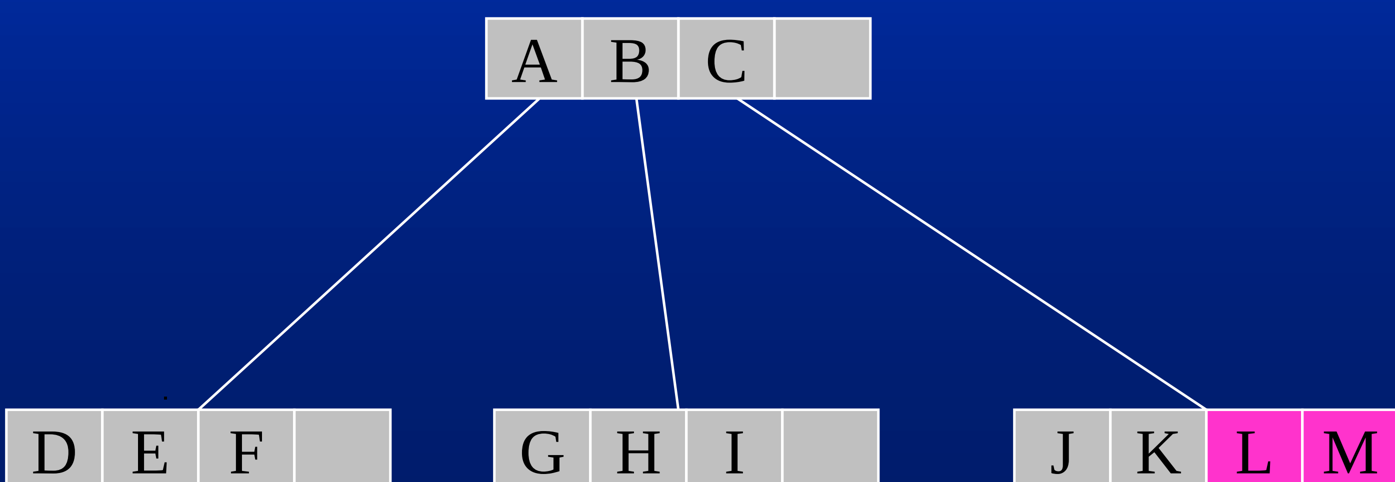
QuadraticSplit

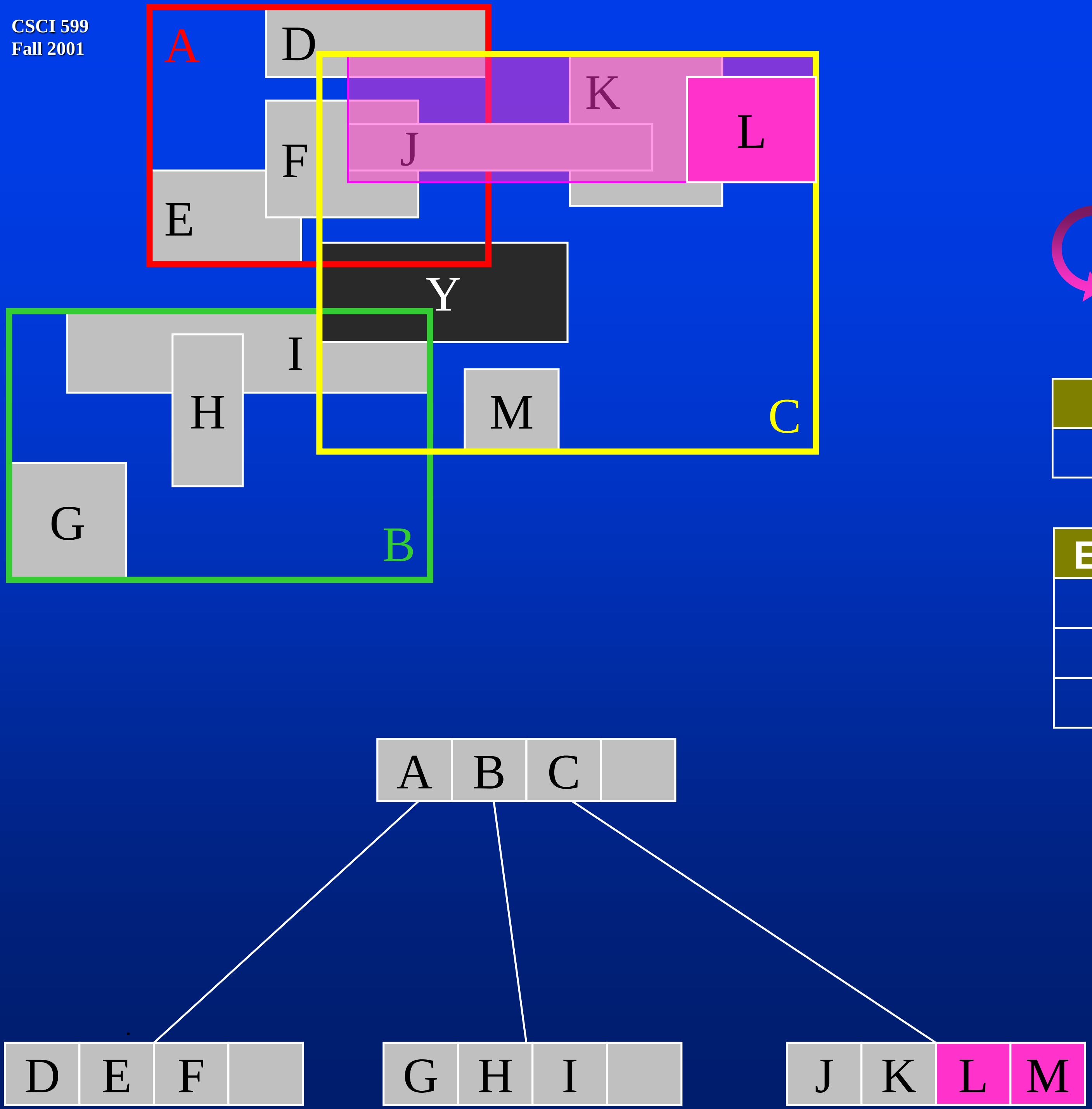
PickSeeds

DistributeEntry



G1	G2
L	M





ChooseSubtree
QuadraticSplit
PickSeeds
DistributeEntry
PickNext

G1	G2
L	M

Entry	d1	d2
J	1.0	
K		
Y		

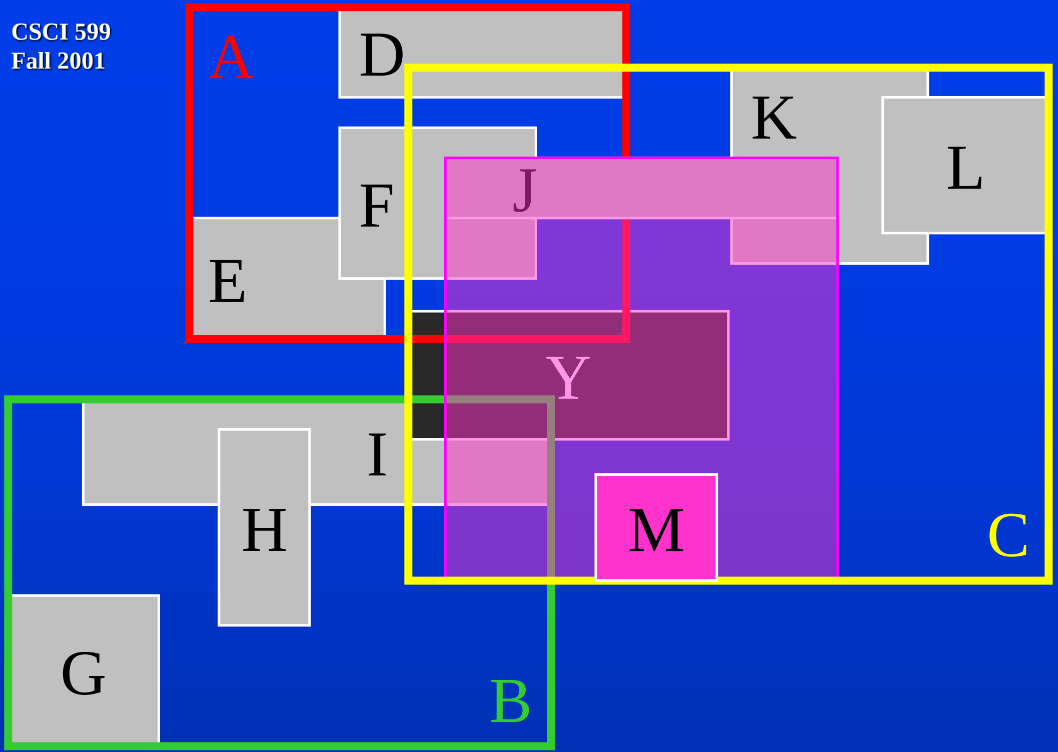
ChooseSubtree

QuadraticSplit

PickSeeds

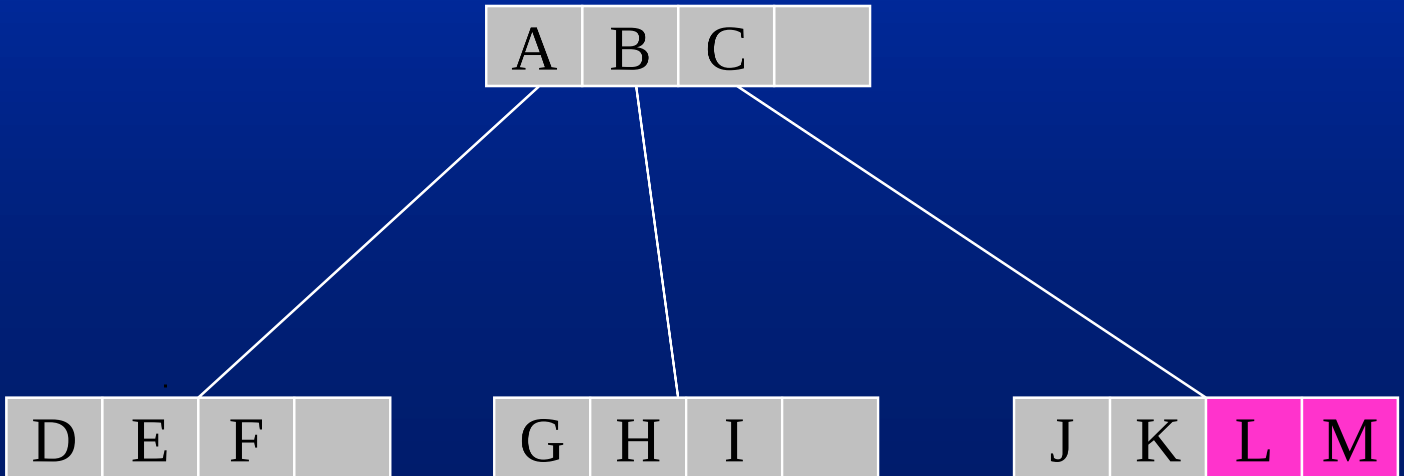
DistributeEntry

PickNext

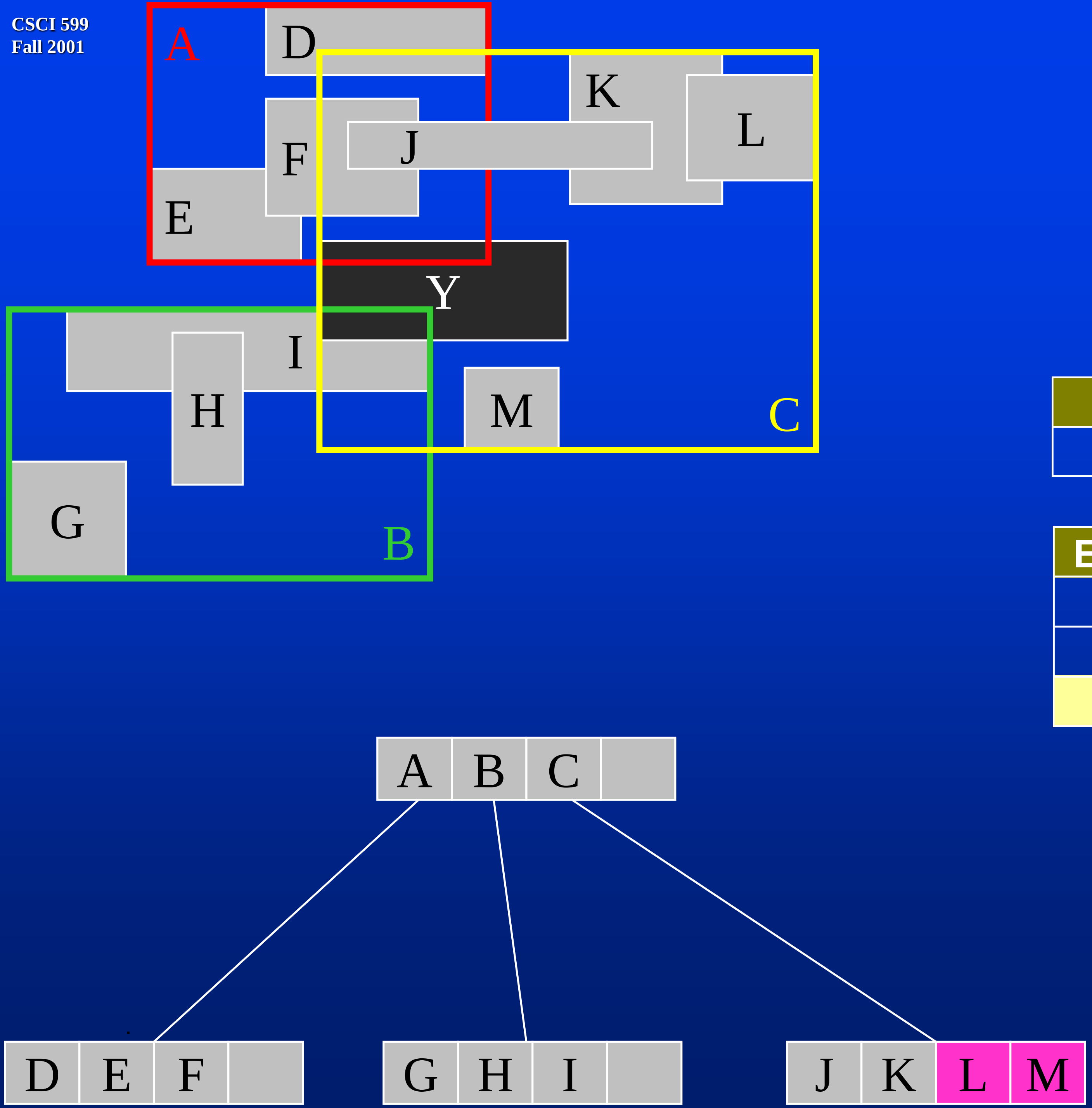


G1	G2
L	M

Entry	d1	d2
J	1.0	2.0
K		
Y		

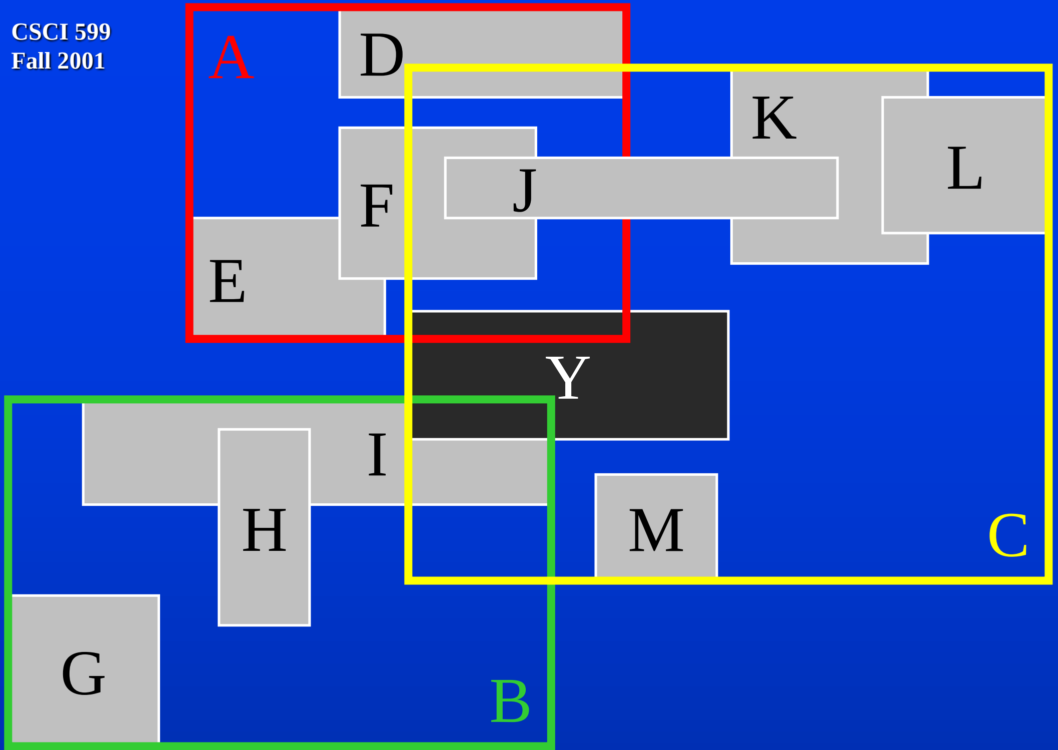


ChooseSubtree
QuadraticSplit
PickSeeds
DistributeEntry
PickNext



G1	G2
L	M

Entry	d1	d2
J	1.0	2.0
K	0.5	2.1
Y	2.6	0.9



ChooseSubtree

QuadraticSplit

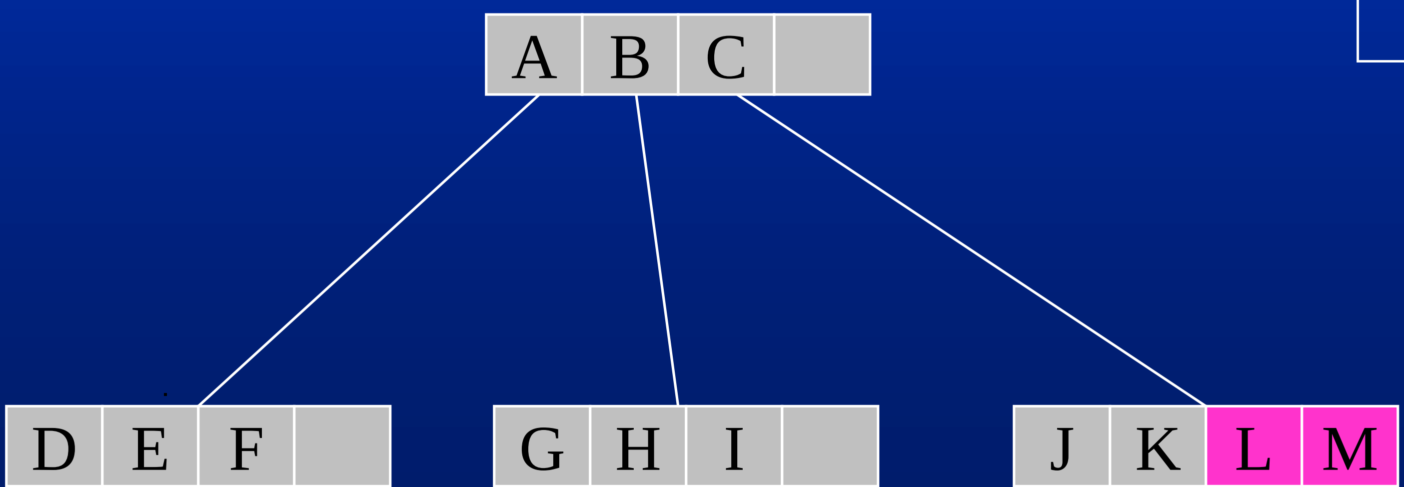
PickSeeds

DistributeEntry

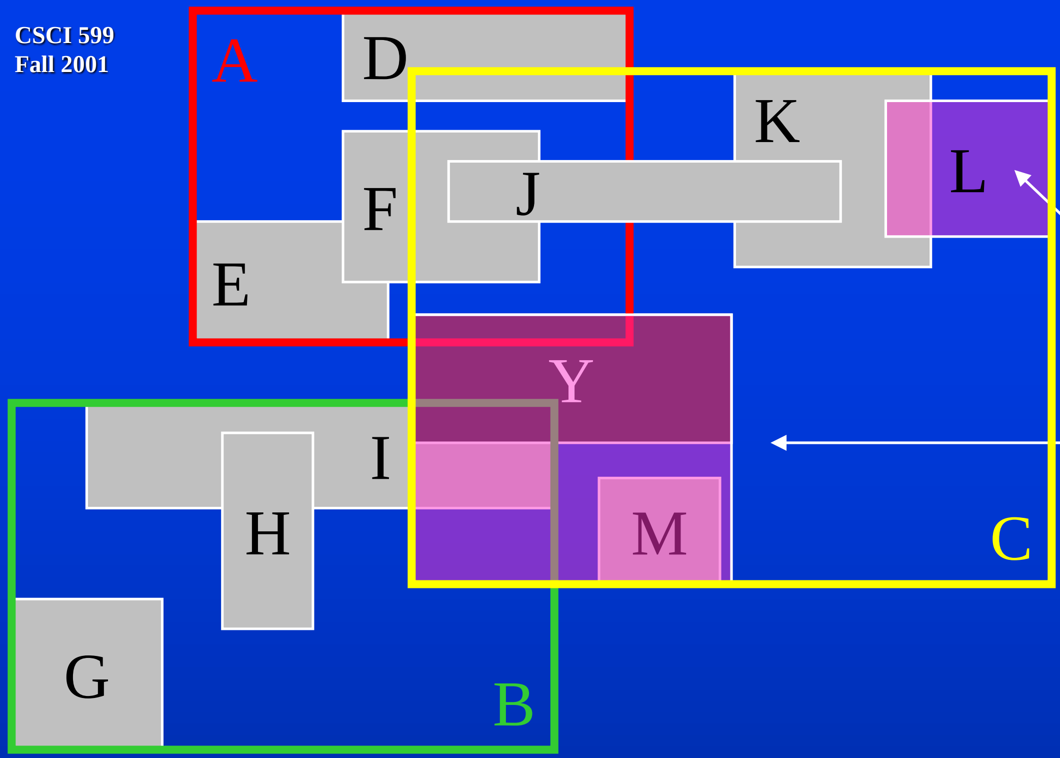
PickNext

G1	G2
L	M
	Y

Entry	d1	d2
J	1.0	2.0
K	0.5	2.1

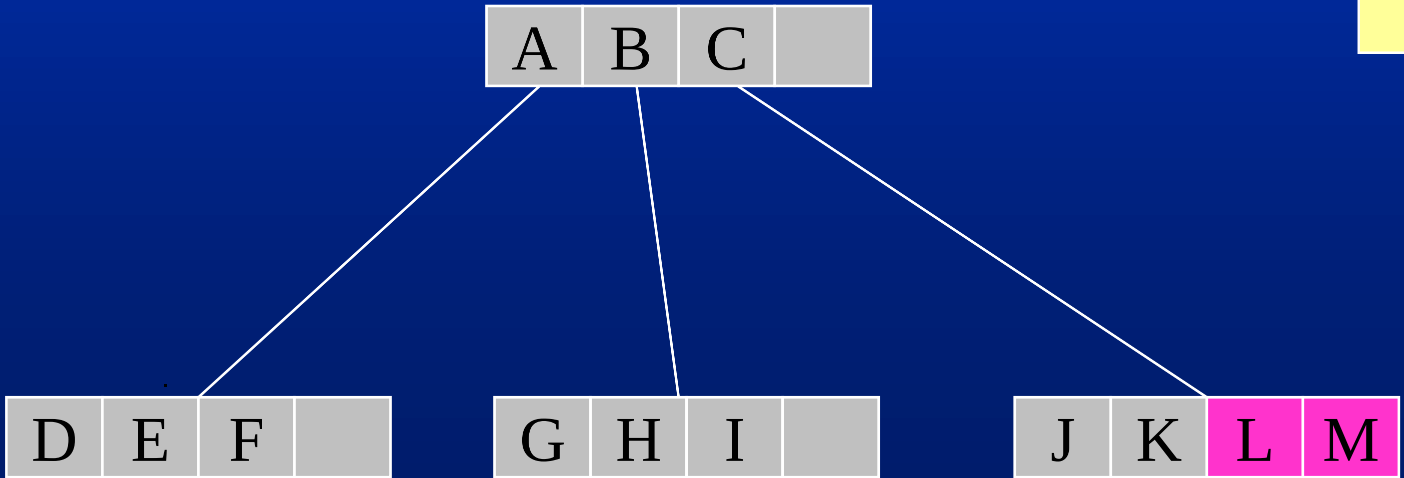


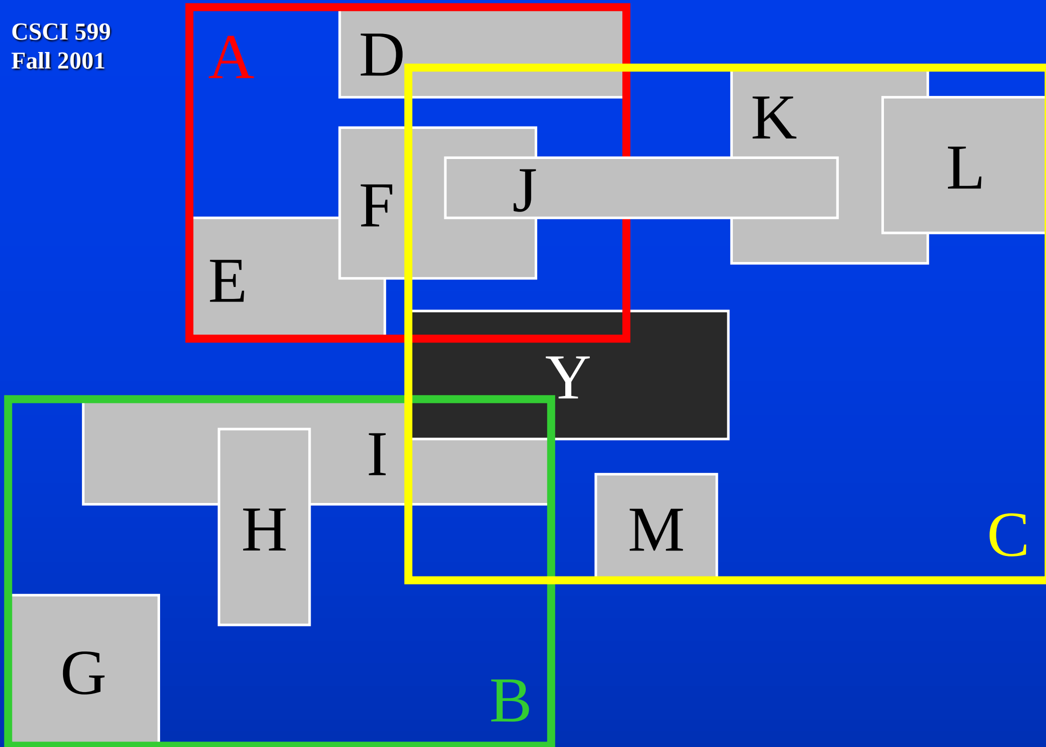
ChooseSubtree
QuadraticSplit
PickSeeds
DistributeEntry
PickNext



G1	G2
L	M
	Y

Entry	d1	d2
J	1.0	1.2
K	0.5	2.3

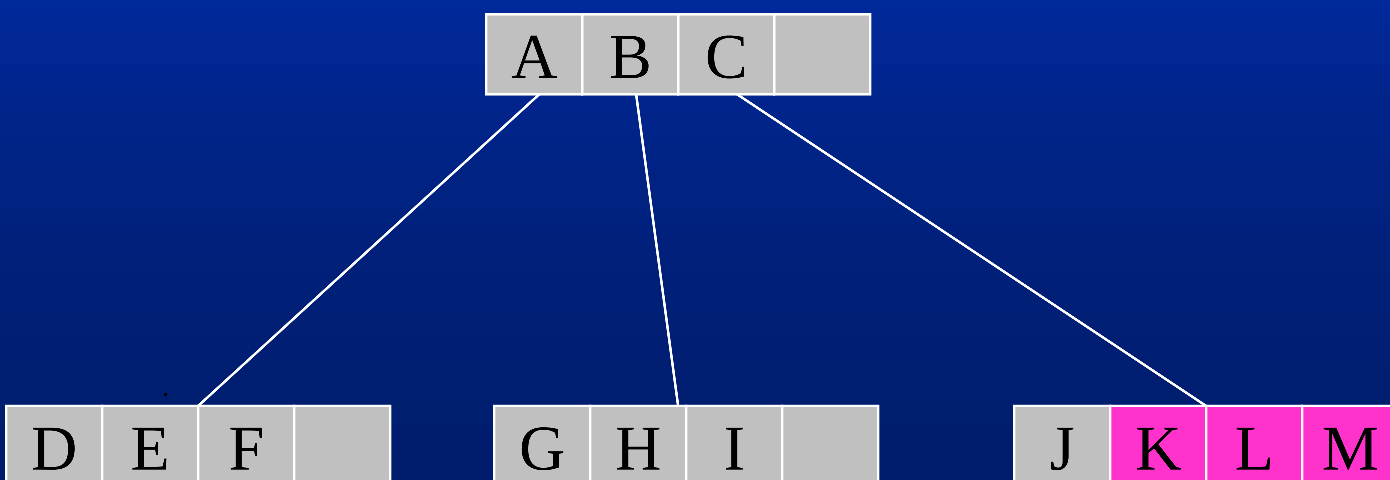


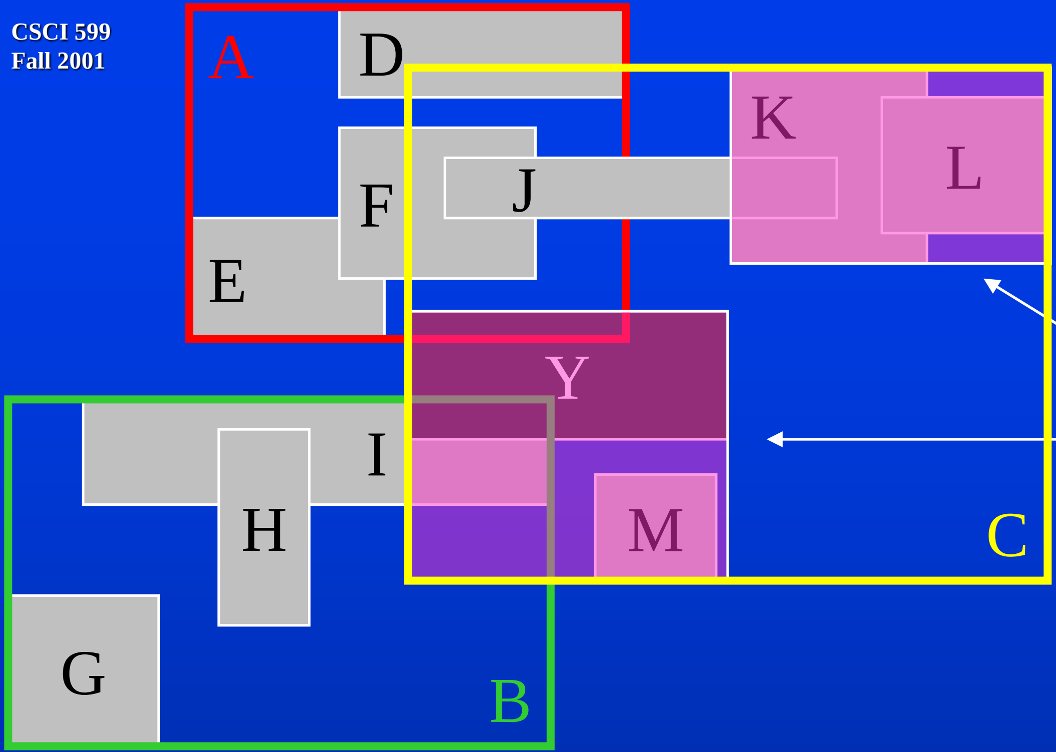


ChooseSubtree
QuadraticSplit
PickSeeds
DistributeEntry
PickNext

G1	G2
L	M
K	Y

Entry	d1	d2
J	1.0	1.2

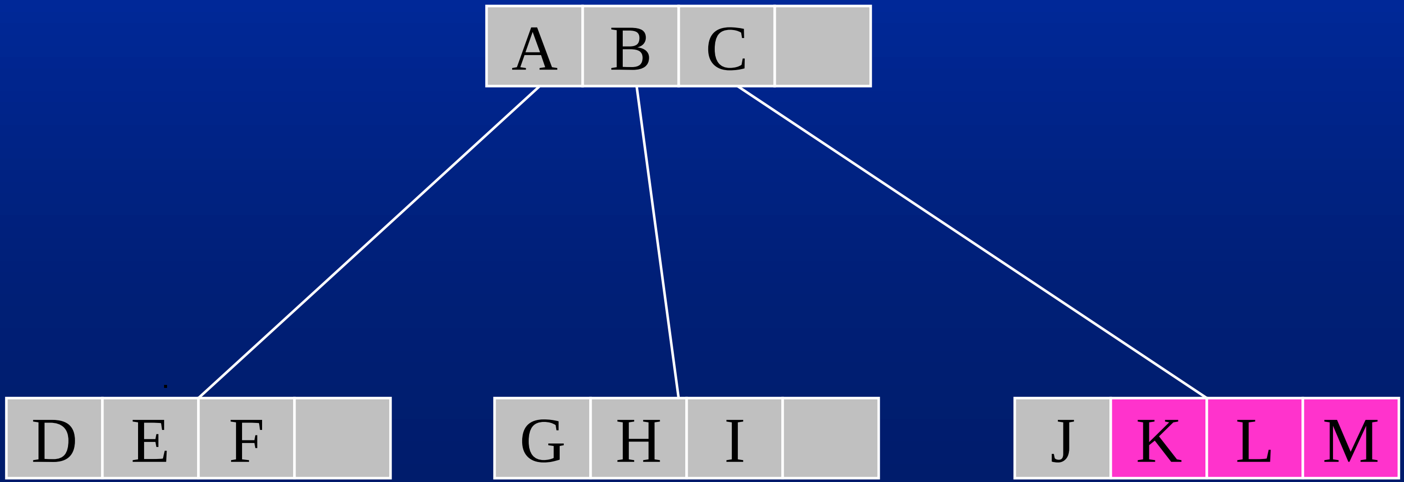


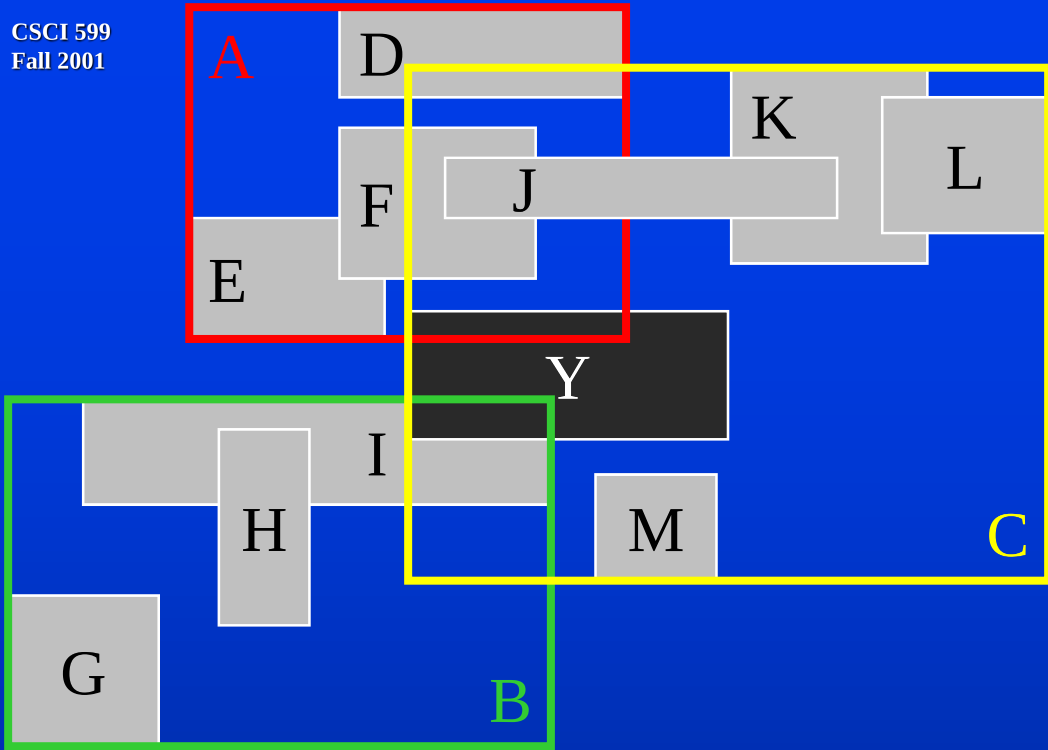


ChooseSubtree
QuadraticSplit
PickSeeds
DistributeEntry
PickNext

G1	G2
L	M
K	Y

Entry	d1	d2
J	0.7	1.2





ChooseSubtree

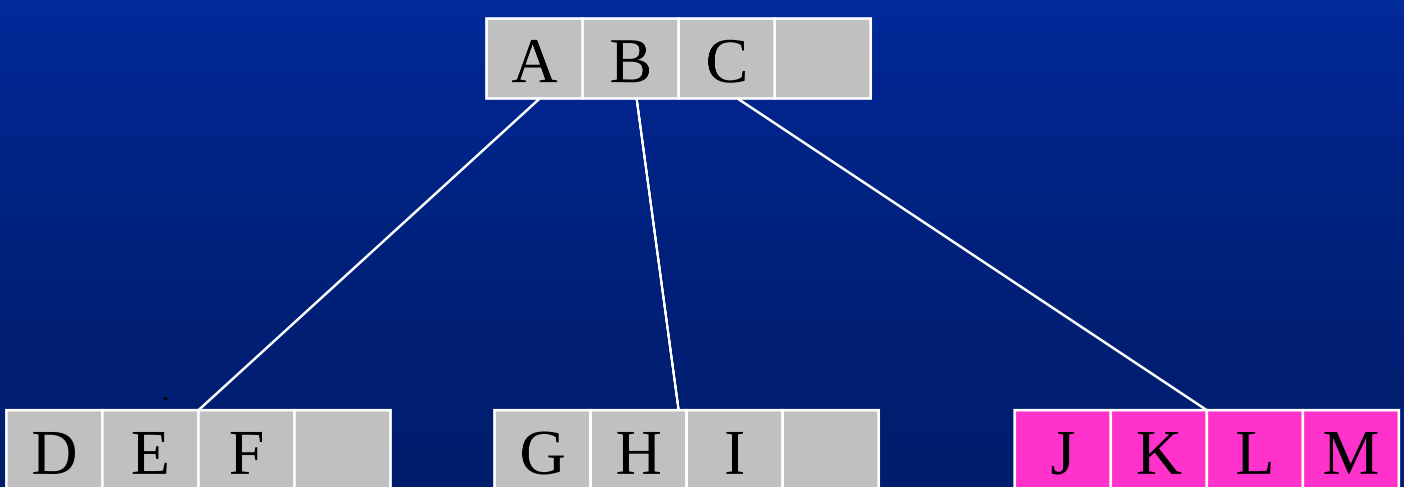
QuadraticSplit

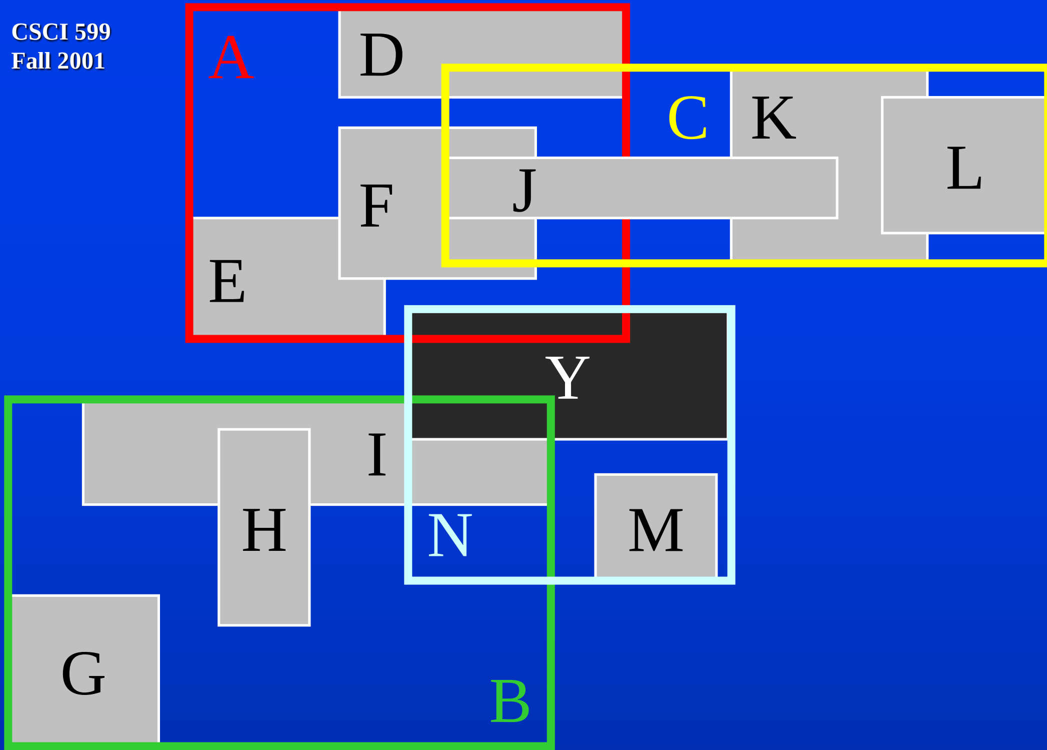
PickSeeds

DistributeEntry

PickNext

G1	G2
L	M
K	Y
J	





ChooseSubtree

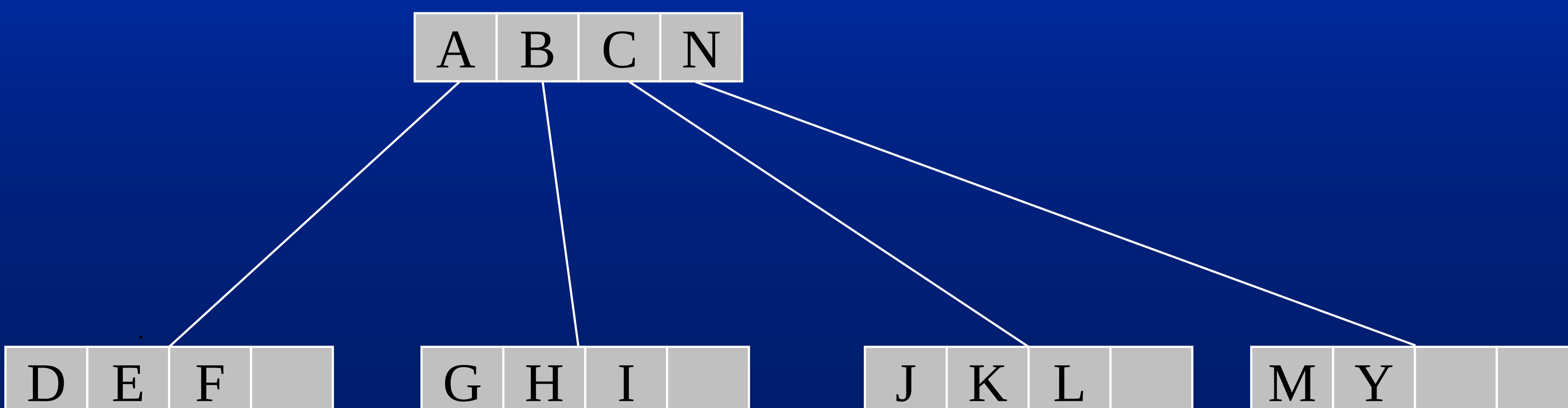
QuadraticSplit

PickSeeds

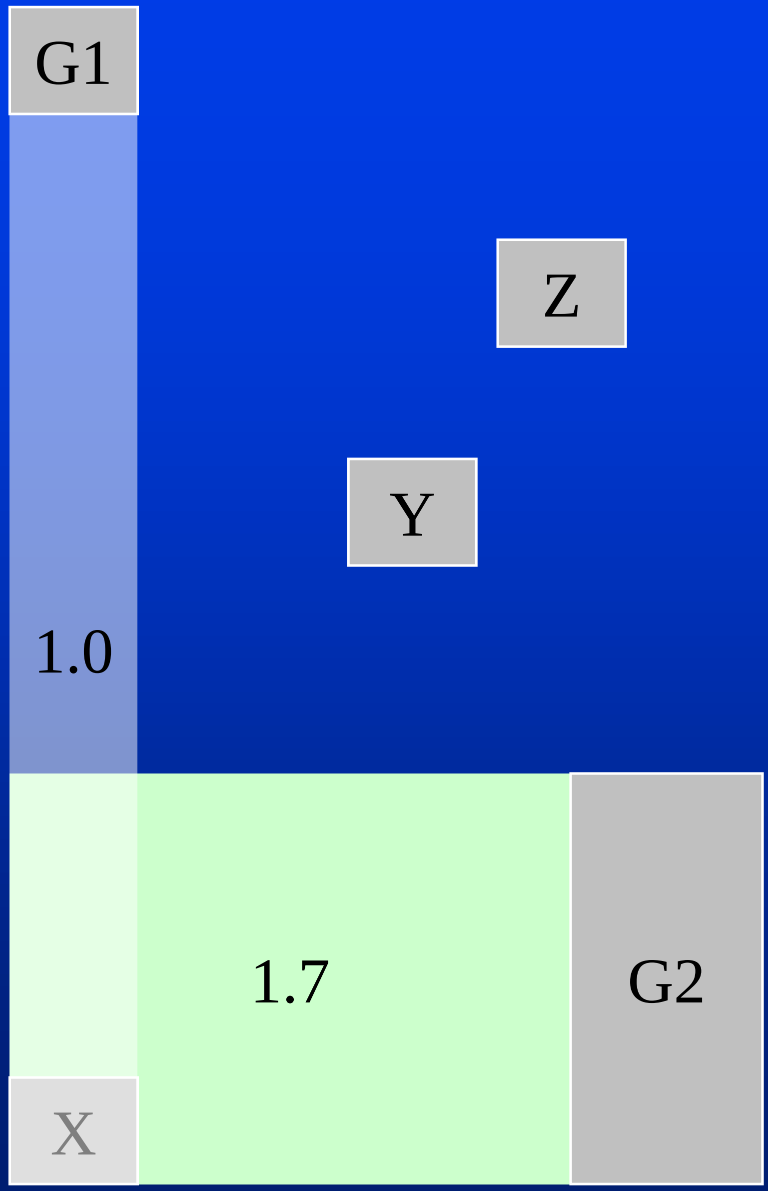
DistributeEntry

PickNext

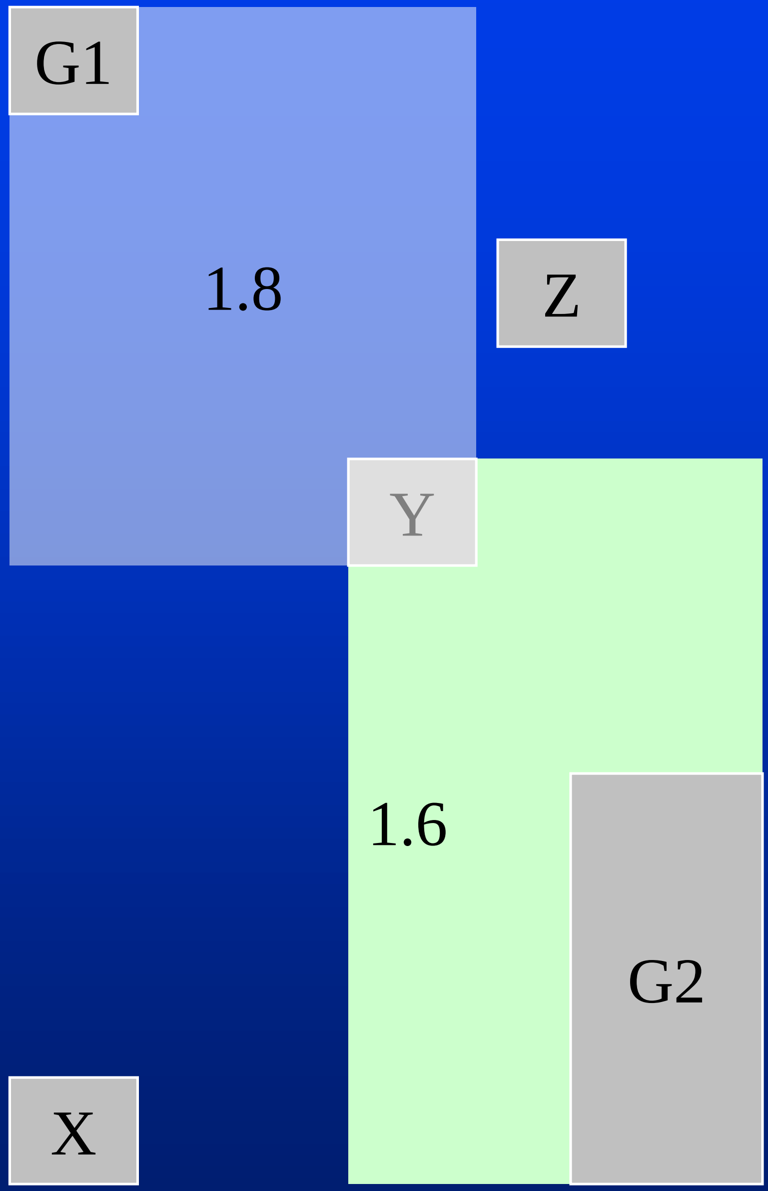
G1	G2
L	M
K	Y
J	



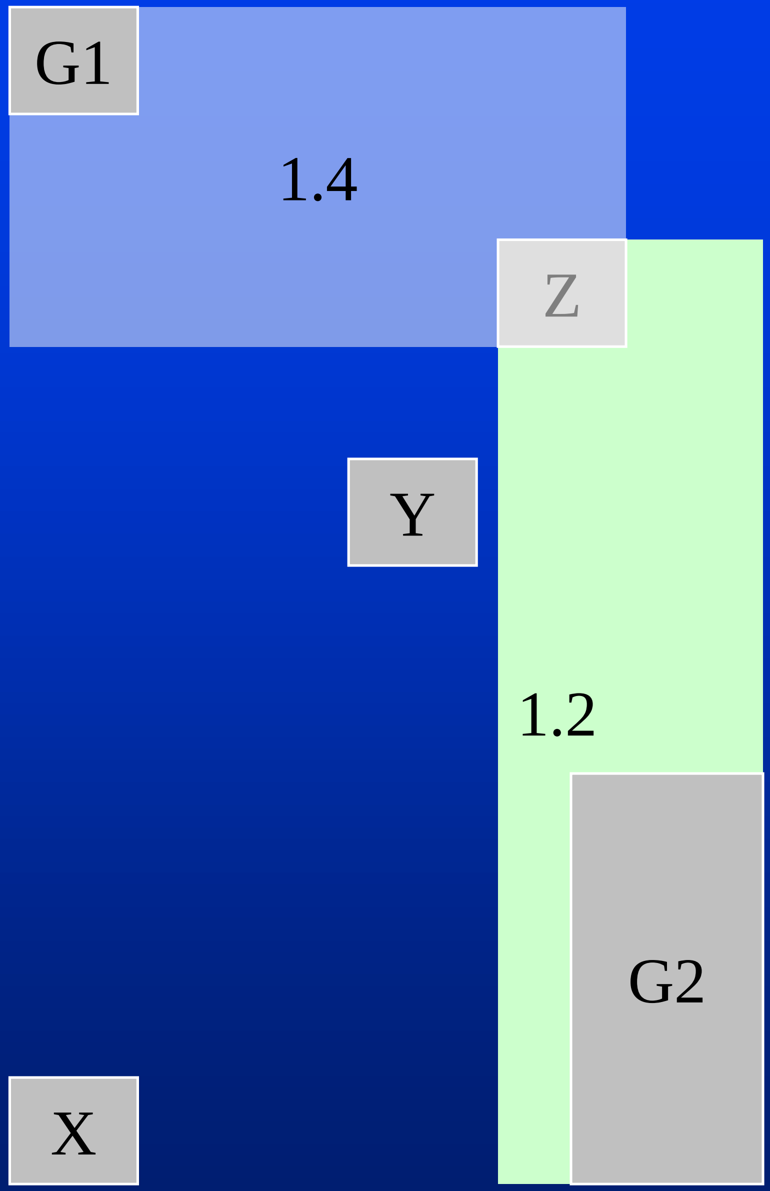
Problem #1: Small Seeds



Problem #1: Small Seeds



Problem #1: Small Seeds



X is assigned to G1. But the distance is large, and hence a bad split. X shall be assigned to G2 instead

Problem #1: Small Seeds

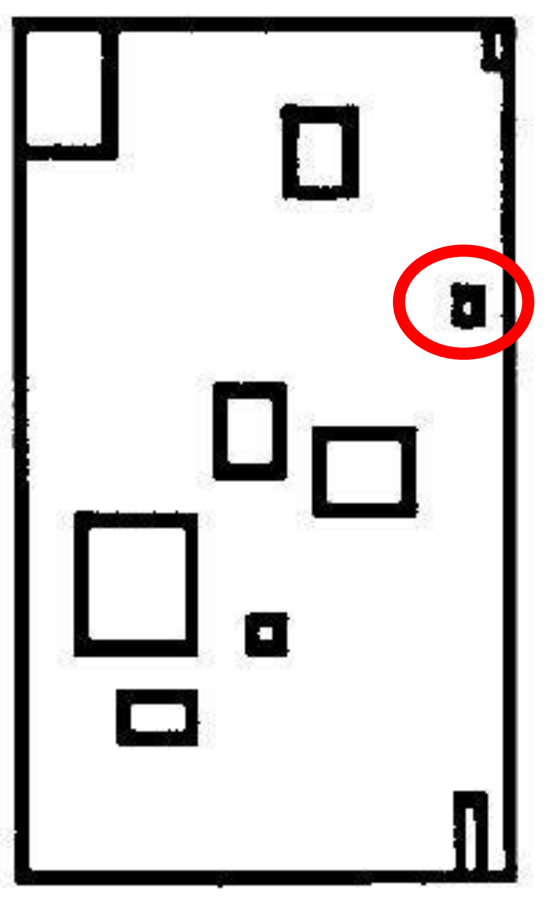


Figure 1a Overfilled node

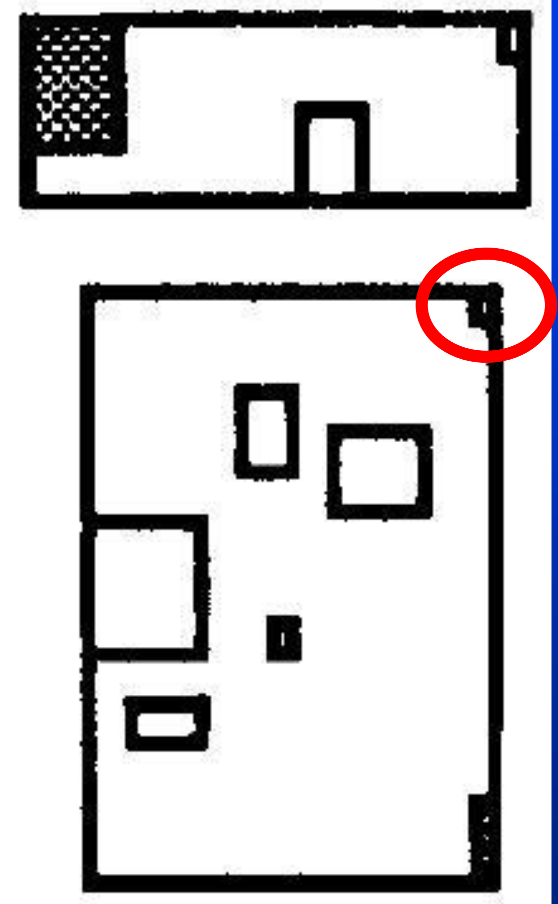


Figure 1b:
Split of the quadratic
R-tree, m = 30%

Problem #2: Prefer Bounding Rectangle

- Assign a rectangle to one seed
- Area enlarged
- Need lesser area enlargement to include next entry
- Enlarge again
- Goes on……

Problem #2: Prefer Bounding Rectangle

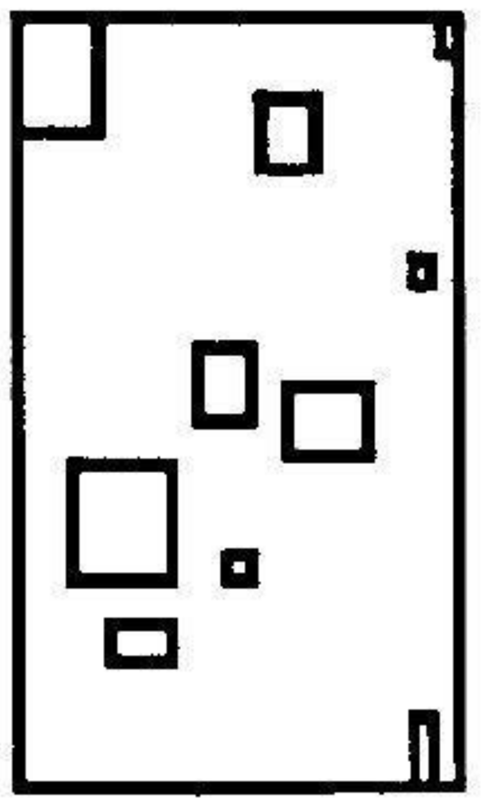


Figure 1a Overfilled node

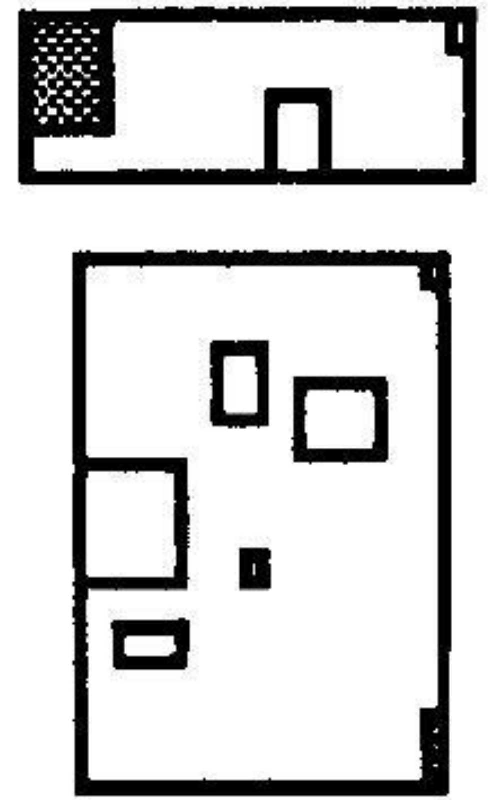


Figure 1b: Split of the quadratic R-tree, m = 30%

M=9
m=3

- M - m + 1 = 7
- Uneven distribution
- Reduce storage utilization

Problem #3

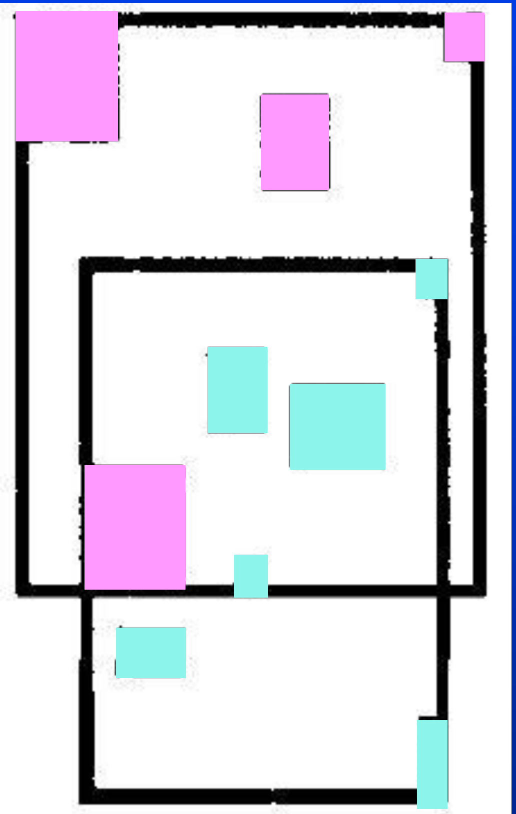


Figure 1c:
Split of the quadratic R-tree,
m = 40%

- If reached M–m+1, all remaining m entries are assigned to the other group without considering geometric properties
- Best retrieval performance when m = 40%

- M - m + 1 = 6
- Large overlap
- Increase number of paths to be traversed

R-tree Variants - Greene

- Alternative split-algorithm
  - Use Guttman's ChooseSubtree algorithm
- The only geometric criterion: choice of split axis
- More geometric optimization criteria needed to improve retrieval performance
- May not find "right" axis and bad split may result

Algorithm Greene's-Split

[Divide a set of M+1 entries into two groups]

GS1 Invoke ChooseAxis to determine the axis perpendicular to which the split is to be performed

GS2 Invoke Distribute

R-tree Variants - Greene

Algorithm ChooseAxis

[Divide a set of M+1 entries into two groups]

CA1 Invoke PickSeeds to find two most distant rectangles of the current node

CA2 For each axis record the separation of the two seeds

CA3 Normalize the separations by dividing them by the length of the nodes enclosing rectangle along the appropriate axis

CA4 Return the axis with the greatest normalized separation

Algorithm Distribute

D1 Sort the entries by the low value of their rectangles along the chosen axis

D2 Assign the first (M+1) / 2 entries to one group, the last (M+1) / 2 entries to the other

D3 If M+1 is odd, then assign the remaining entry to the group whose enclosing rectangle will be increased least by its addition

R-tree Variants - Greene

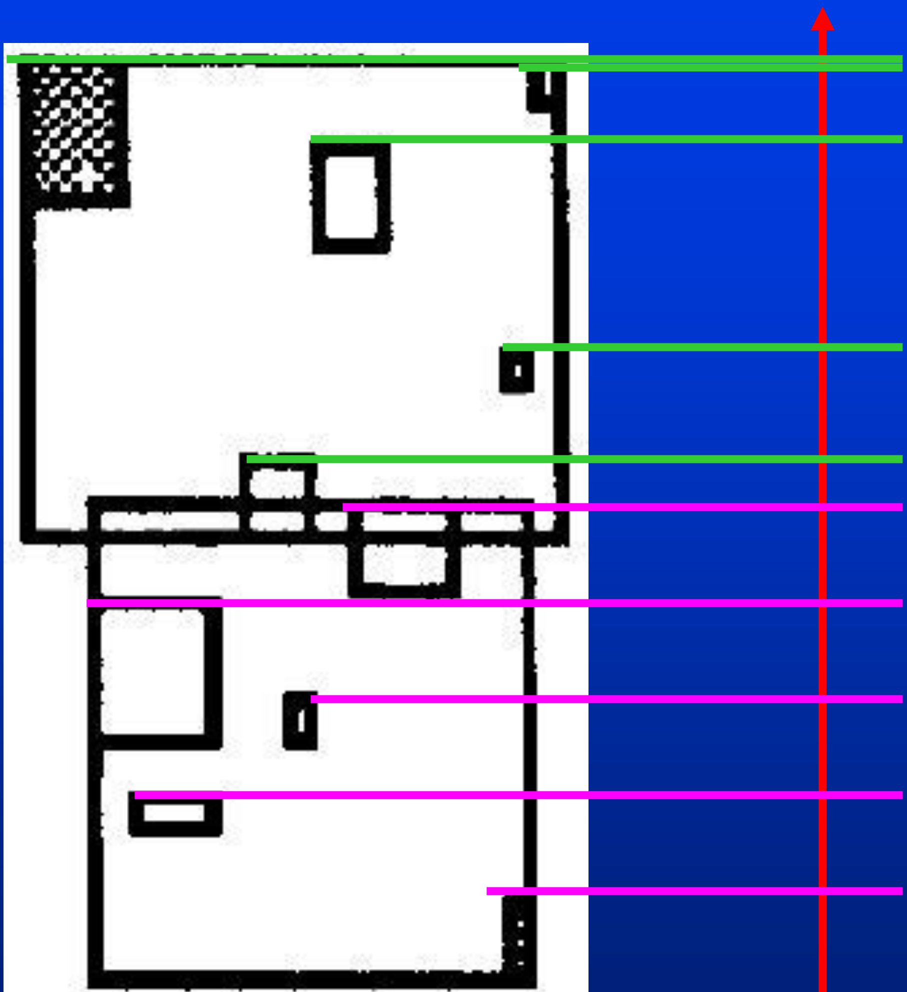


Figure 1d
Greene s split

- M = 9
- (M + 1) div 2 = 5

R*-tree

- Previous R-trees consider only area parameter when choosing insertion path
- R*-tree considers area, margin and overlap in different combinations

Algorithm ChooseSubtree

CS1 Set N to be the root node

CS2 If N is a leaf,

return N

else

If the childpointers in N point to leaves

[determine the minimum overlap cost]

choose the entry in N whose rectangle needs least overlap enlargement to include the new data rectangle. Resolve ties by choosing the entry whose rectangle needs least area enlargement, then the entry with the rectangle of smallest area

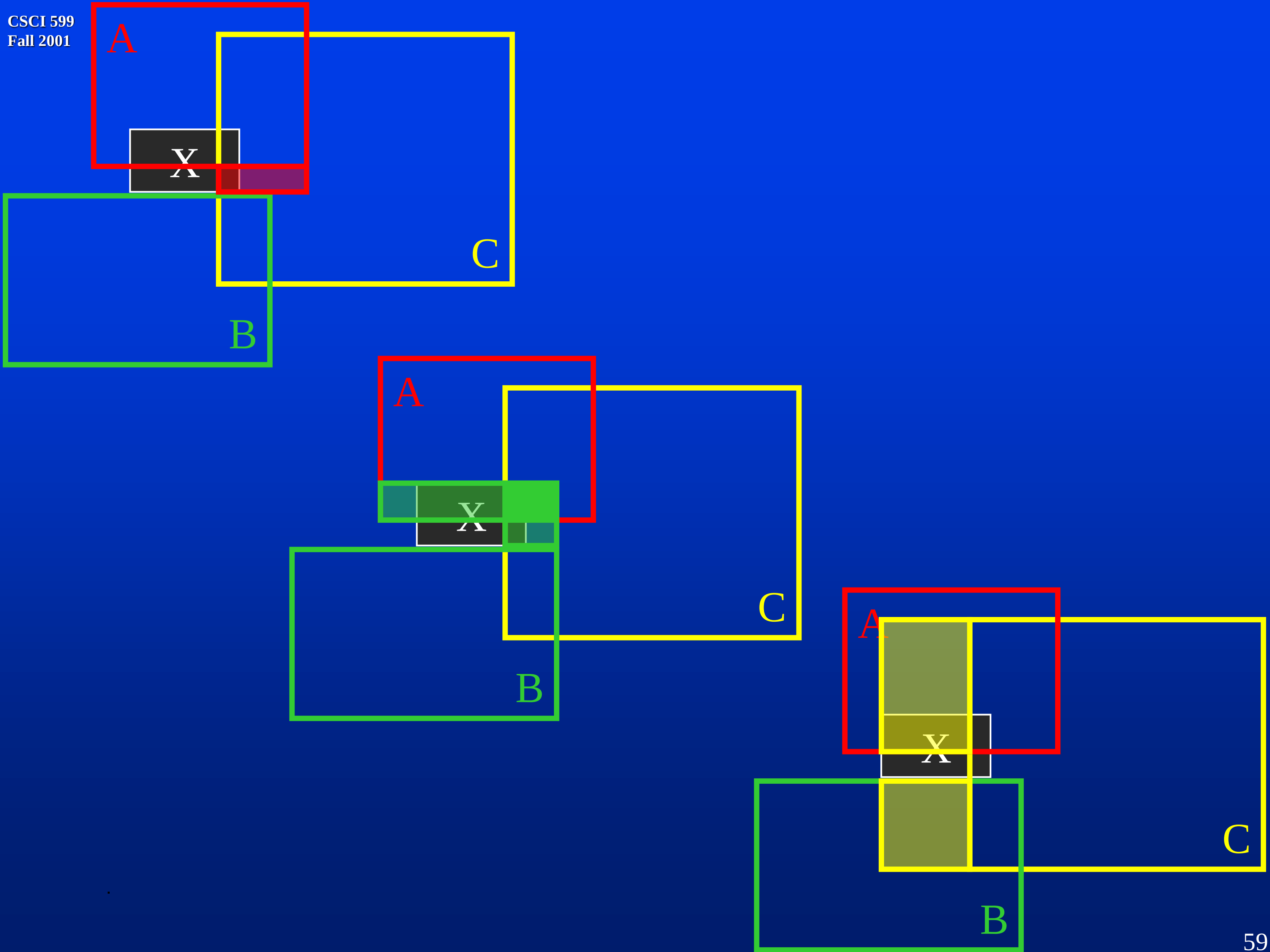
else

[determine the minimum area cost]

choose the entry in N whose rectangle needs least area enlargement to include the new data rectangle. Resolve ties by choosing entry with rectangle of smallest area

end

CS3 Set N to be the childnode pointed to by the childpointer of the chosen entry. Repeat from CS2


A
X
C
B
A
X
C
B
A
X
C
B

Algorithm ChooseSubtree

- For choosing best leaf node, minimizing overlap performed slightly better (smaller overlap means smaller # of paths and hence higher performance)
- Cpu cost of determining overlap is quadratic in the number of entries
- For large node sizes, reduce calculation by modifying:

  [determine the nearly minimum overlap cost]

  Sort the rectangles in N in increasing order of their area enlargement needed to include the new data rectangle.

  Let A be the group of the first p entries. From the entries in A, considering all entries in N, choose the entry whose rectangle needs least overlap enlargement. Resolve ties as described before

Algorithm ChooseSubtree

- For 2-D, p = 32: nearly no reduction of retrieval performance compared to unmodified
- Cpu cost remains higher than original ChooseSubtree
- Improve retrieval performance particularly in queries with small query rectangles on datafiles with non-uniformly distributed small rectangles or points
- Improve robustness

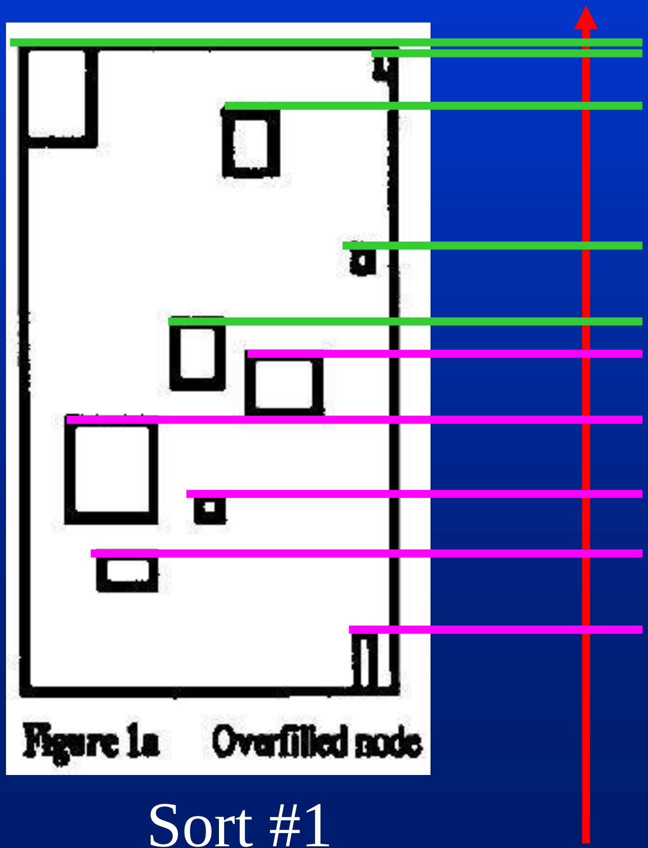
Split of R*-tree

- **Along each axis**
  - **entries first sorted by lower value of their rectangles, then sorted by upper value**
  - **determine (M–2m+2) distributions of (M+1) entries into two groups for each sort**

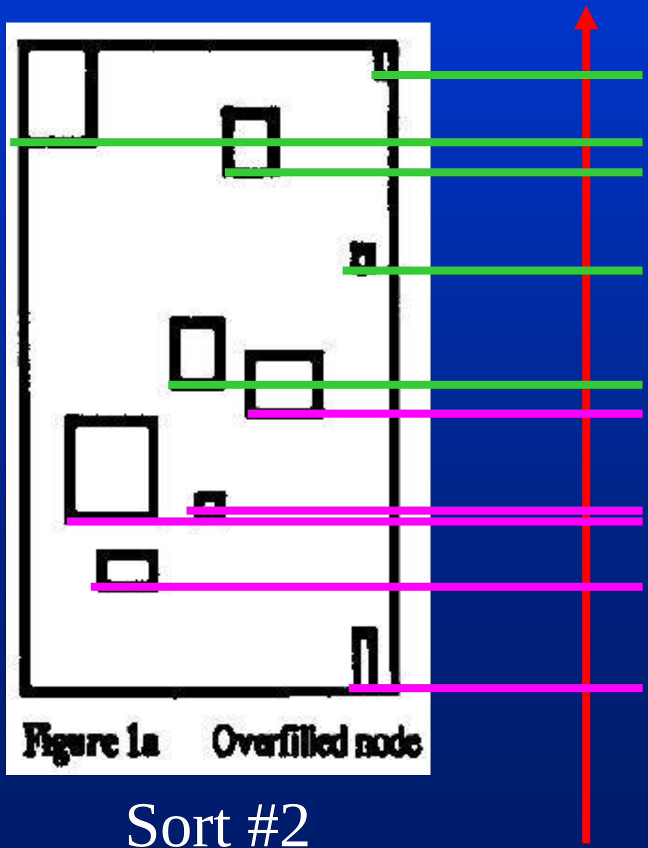
Distribution #	Group 1	Group 2
1	m	M - m + 1
2	m + 1	M - m
3	m + 2	M - m - 1
……..	……..	………
M - 2m + 2	M - m + 1	m

e.g. M = 9, m = 4, 10 entries

Distribution #	Group 1	Group 2
1	4	6
2	5	5
3	6	4



Sort #1



Sort #2

Split of R*-tree

- Determine goodness values for each distribution
  - three goodness values
    - area-value area[bb(first group)] + area[bb(second group)]
    - margin-value margin[bb(first group)] + margin[bb(second group)]
    - overlap-value area[bb(first group) ∩ bb(second group)]
  - depend on goodness values, final distribution is determined

bb: bounding box of a set of rectangles

Algorithm Split

S1 Invoke ChooseSplitAxis to determine the axis, perpendicular to which the split is performed

S2 Invoke ChooseSplitIndex to determine the best distribution into two groups along that axis

S3 Distribute the entries into two groups

Algorithm ChooseSplitAxis

CSA1 For each axis

Sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above. Compute S, the sum of all margin-values of the different distributions

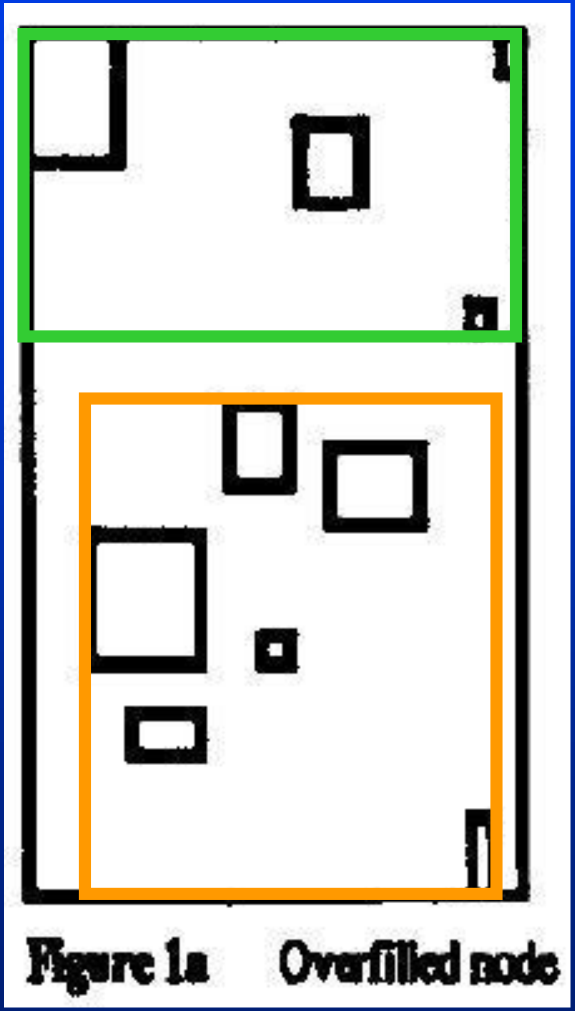
end

CSA2 Choose the axis with the minimum S as split axis

Algorithm ChooseSplitIndex

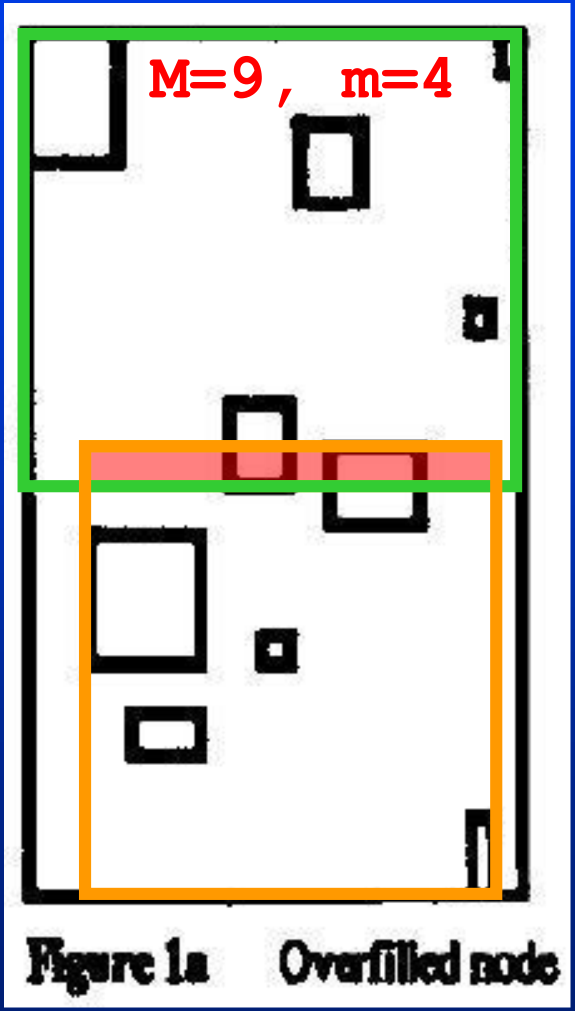
CSA1 Along the chosen split axis, choose the distribution with the minimum overlap-value. Resolve ties by choosing the distribution with minimum area-value

Distribution #1

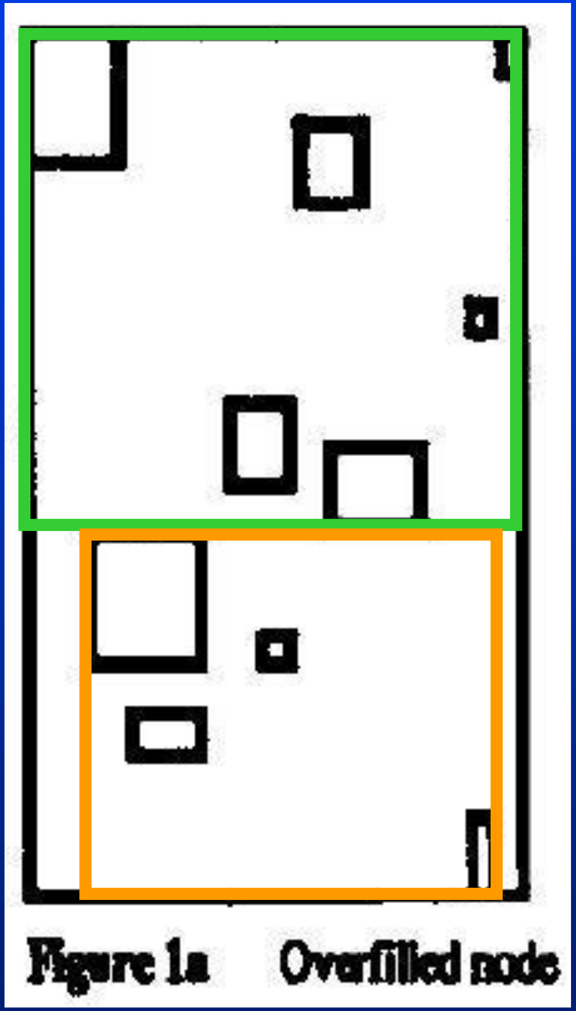


Area-value = 1.00

Distribution #2



Distribution #3



Area-value = 1.11

Split of R*-tree

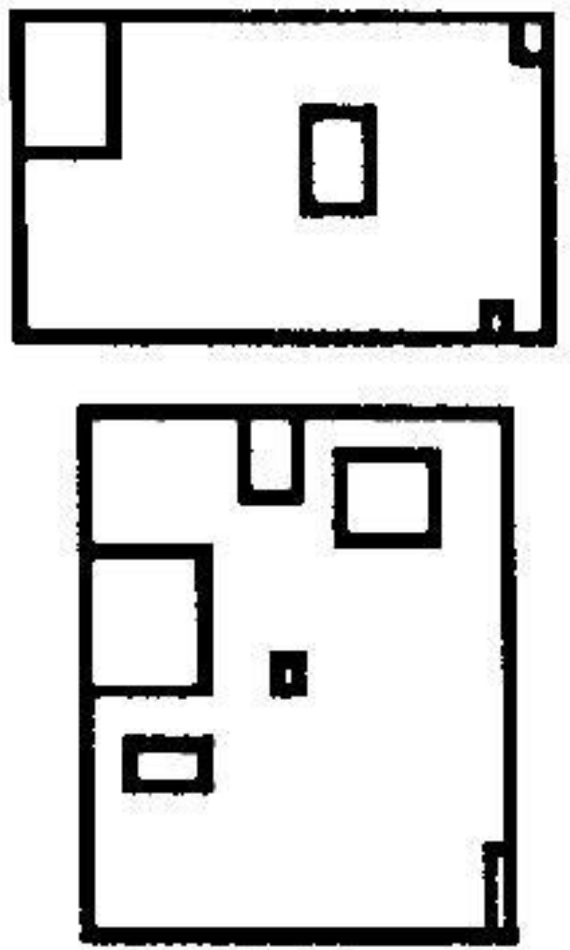


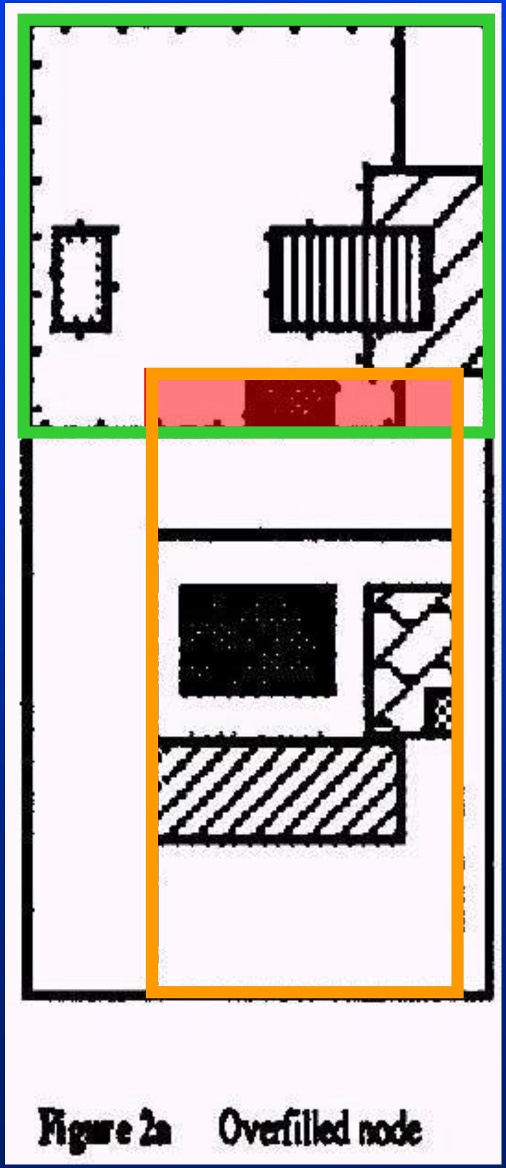
Figure 1e M=9, m=4

Split of the R*-tree, m = 40%

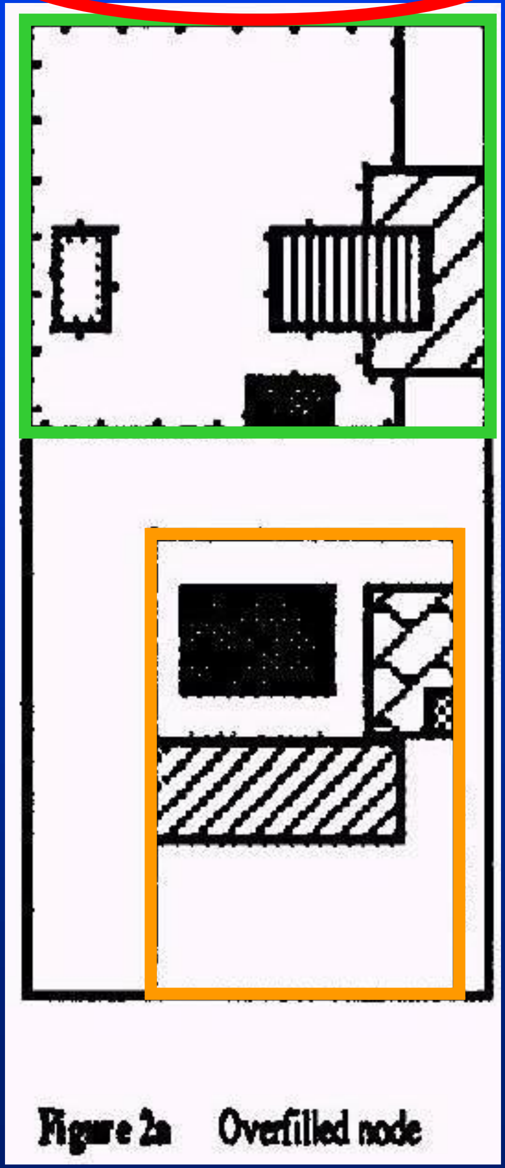
- Tests with m = 20%, 302, 40% and 45% of M
- Best performance when m = 40%

Another Example

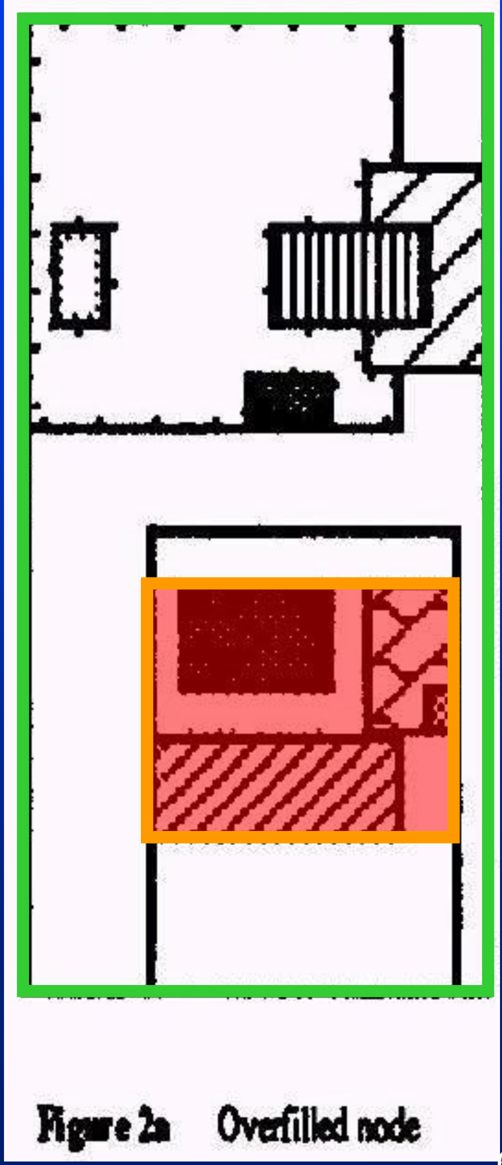
Distribution #1



Distribution #2



Distribution #3



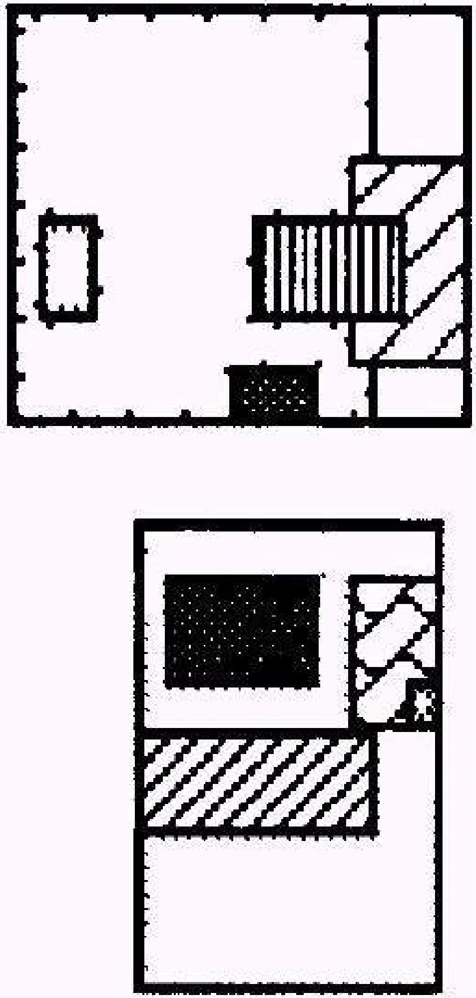


Figure 2c: Split of the R* tree where the splitaxis is vertical

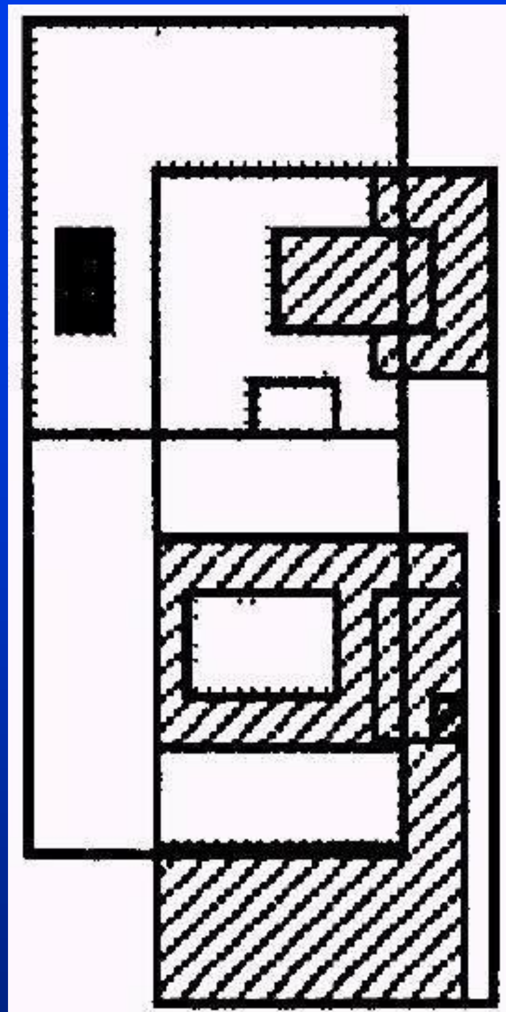


Figure 2b: Greene's split where the split axis is horizontal

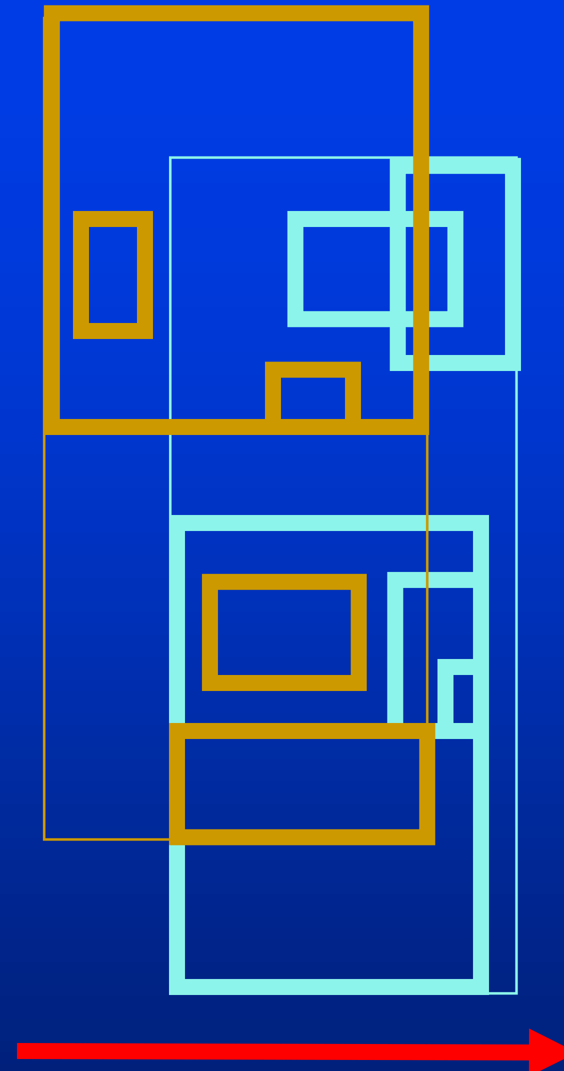


Figure 2b: Greene's split where the split axis is horizontal

Forced Reinsert

- R-tree and R*-tree are nondeterministic
  - different sequences of insertions will build up different trees
  - R-tree suffers from its old entries
- A very local reorganization of directory rectangles is performed during a split
  - it is rather poor
  - need a more powerful and less local instrument to reorganize the structure

Deletion and Insert Algorithms by Guttman

Algorithm Deletion

D1 Invoke Findleaf

D2 Remove index record E

D3 Invoke CondenseTree -------> Invoke Insert

D4 If necessary, make the child the new roof

Algorithm Insert

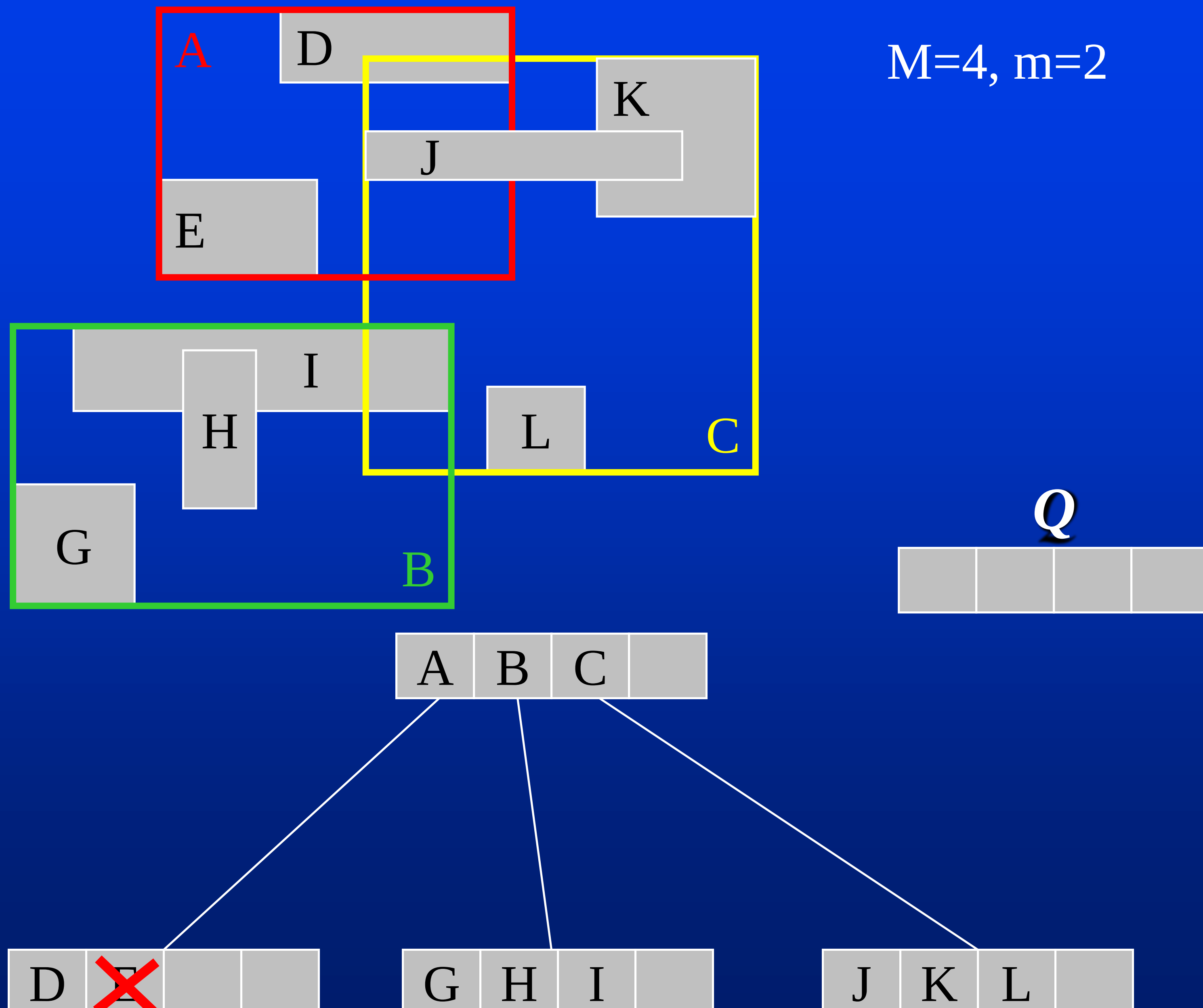
I1 Invoke ChooseSubtree

I2 Install E, or invoke SplitNode (QuadraticSplit)

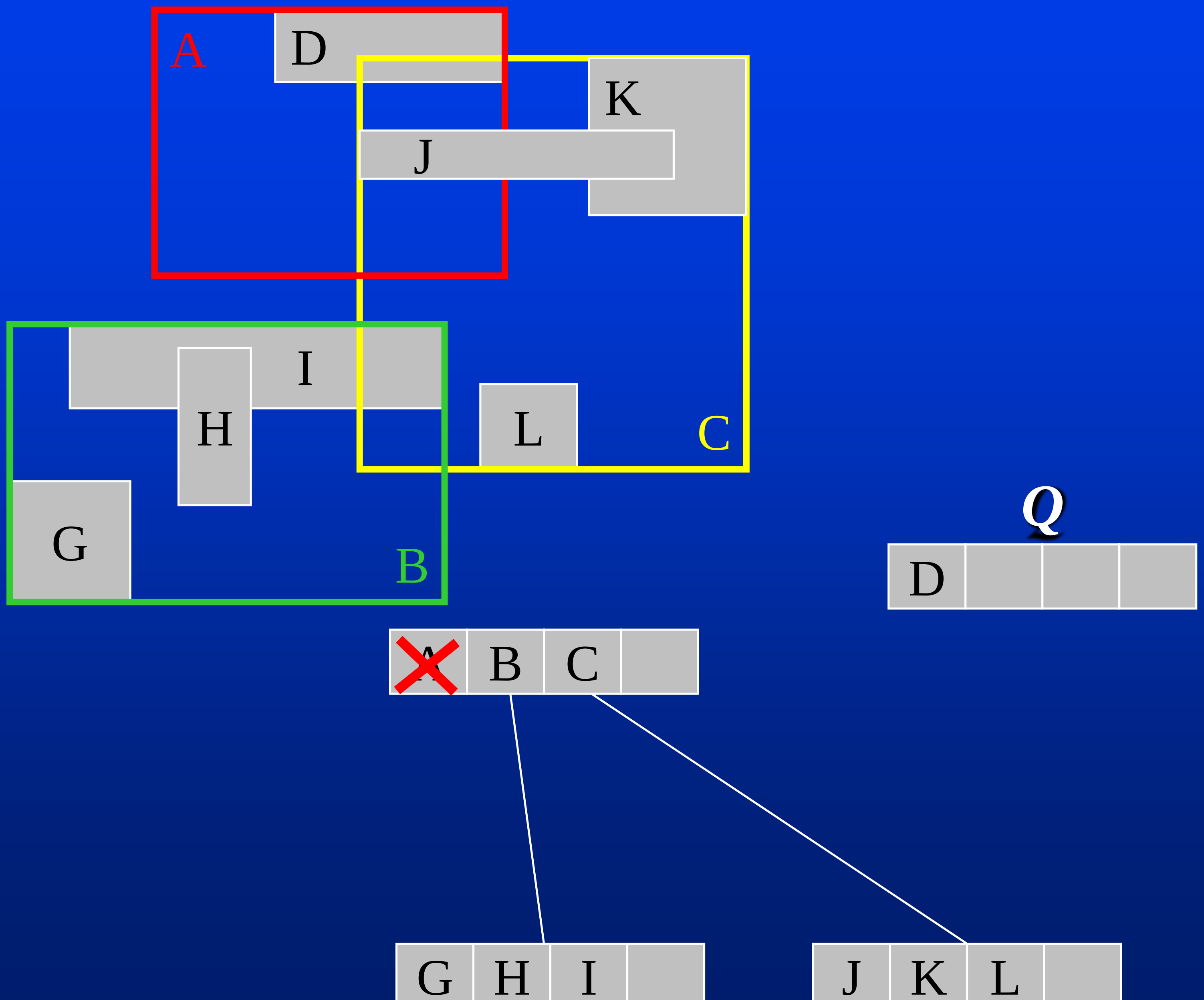
I3 Invoke AdjustTree

I4 If necessary, create new root

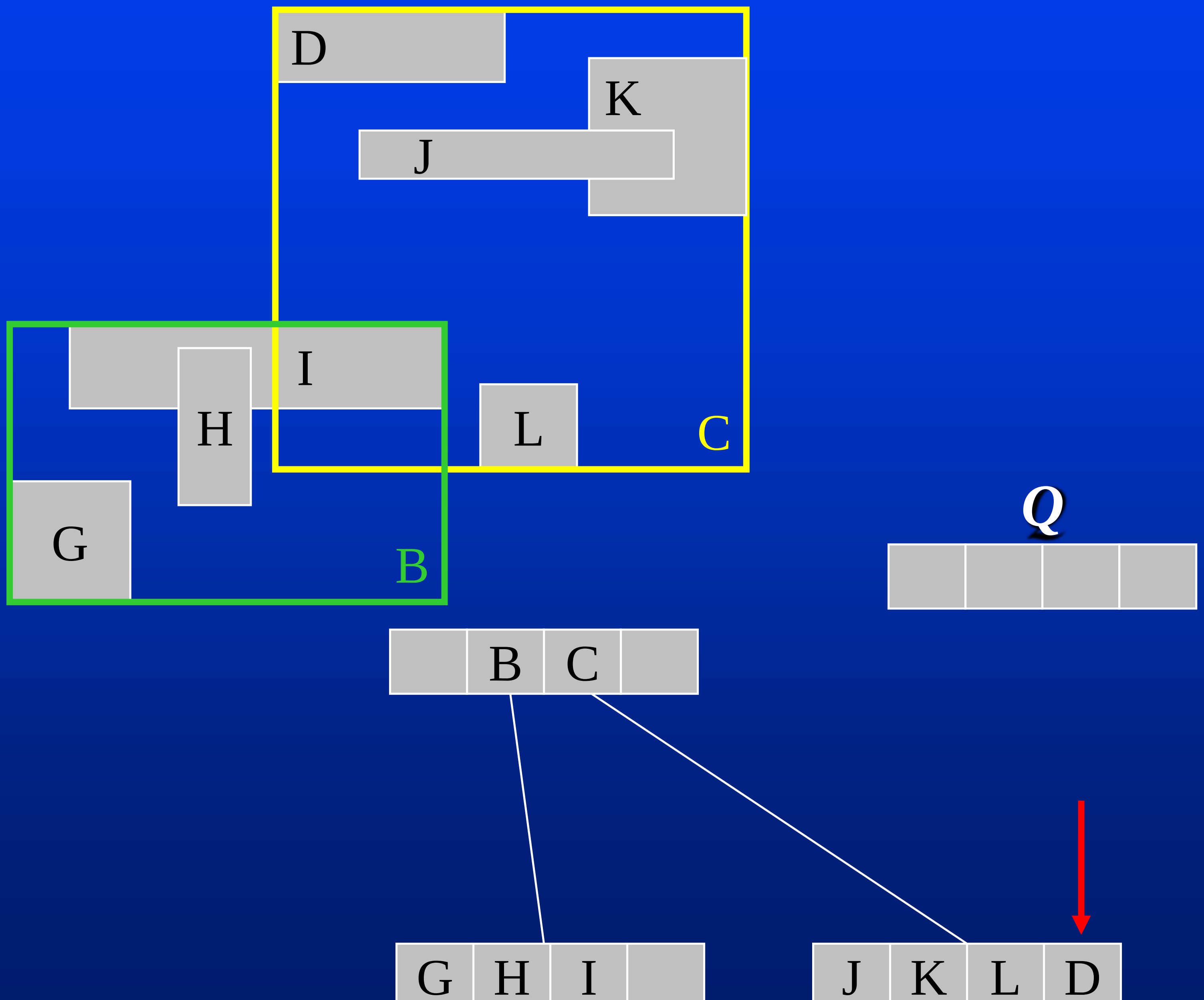
Treating Underfilled Nodes in R-tree



Treating Underfilled Nodes in R-tree



Treating Underfilled Nodes in R-tree



Forced Reinsert

- **Expected that deletion and reinsertion of old data rectangles improve retrieval performance**
  - **experiment showed performance improvement of 20% ~ 50% (static situation)**
- **To achieve dynamic reorganizations, R*-tree forces entries to be reinserted during insertion routines**
  - **algorithms same as Guttman, except for overflow treatment**

Forced Reinsert

Algorithm InsertData

ID1 Invoke Insert starting with the leaf level as a parameter, to insert a new data rectangle

Algorithm Insert

I1 Invoke ChooseSubtree, with the level as a parameter, to find an appropriate node N, in which to place the new entry E

I2 If N has less than M entries, accommodate E in N. If N has M entries, invoke OverflowTreatment with the level of N as a parameter [for reinsertion or split]

I3 If OverflowTreatment was called and a split was performed, propagate OverflowTreatment upwards if necessary

If OverflowTreatment caused a split of the root, create a new root

I4 Adjust all covering rectangles in the insertion path such that they are MBRs enclosing then children rectangles

Algorithm OverflowTreatment

OT1 If the level is not the root level and this is the first call of OverflowTreatment in the given level during the insertion of one data rectangle, then

invoke Reinsert

else

invoke Split

end

Algorithm Reinsert

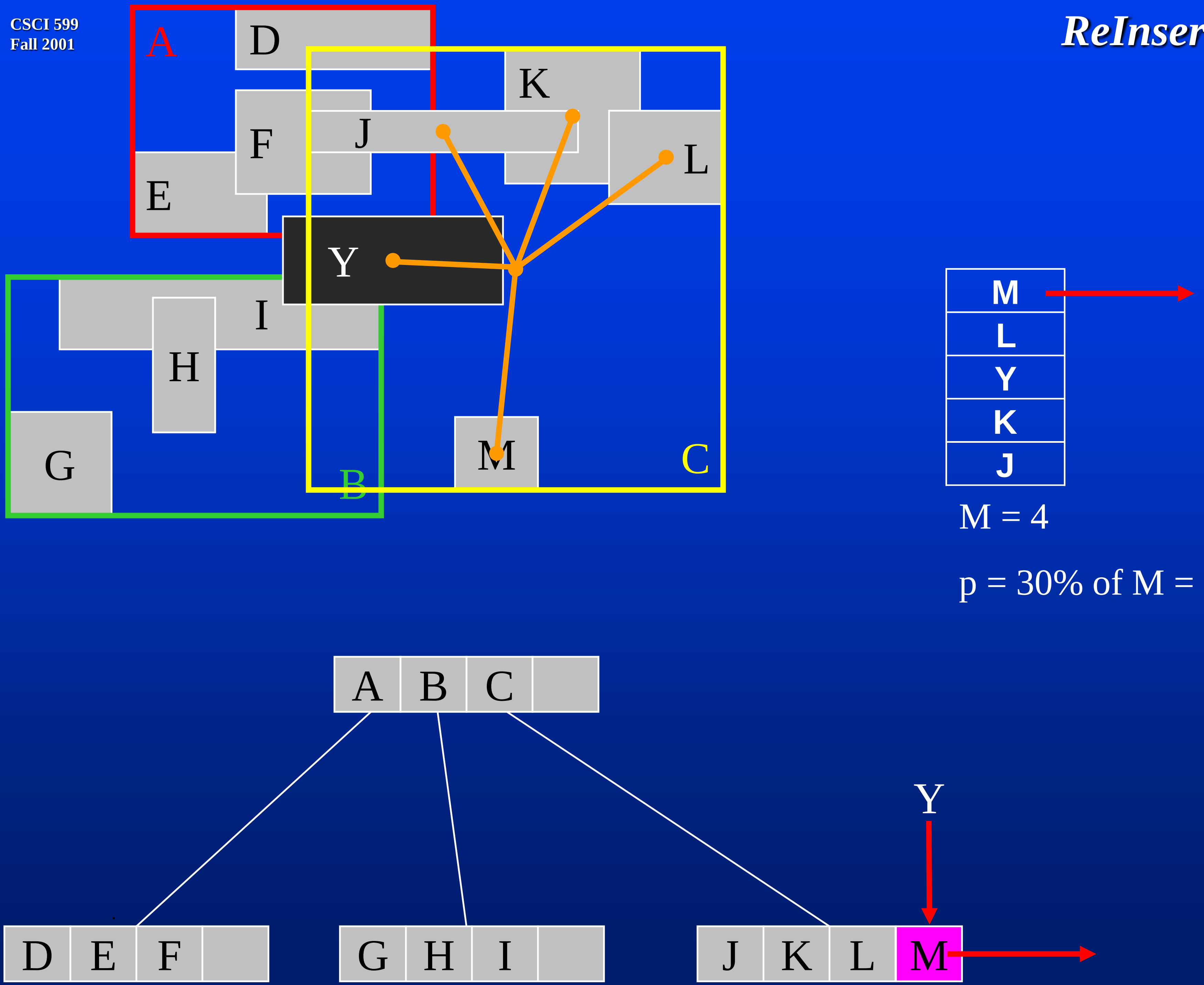
RI1 For all M+1 entries of a node N, compute the distance between the centers of their rectangles and the center of the bounding rectangle of N

RI2 Sort the entries in decreasing order of their distances computed in RI1

RI3 Remove the first p entries from N and adjust the bounding rectangle of N

RI4 In the sort, defined in RI2, starting with the maximum distance (= far reinsert) or minimum distance (= close reinsert), invoke Insert to reinsert the entries

ReInsert



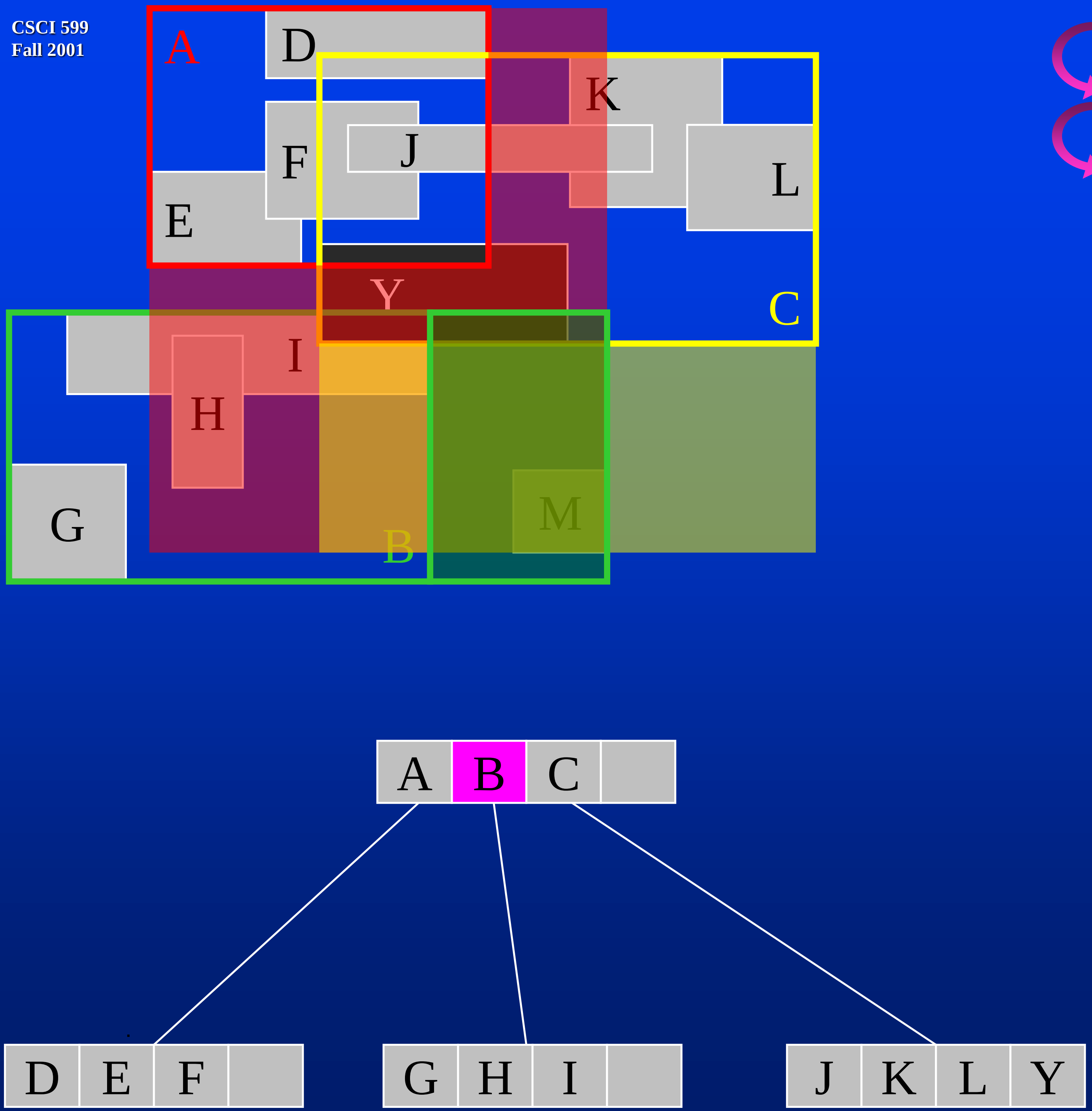
M = 4

p = 30% of M = 1

ReInsert

Insert

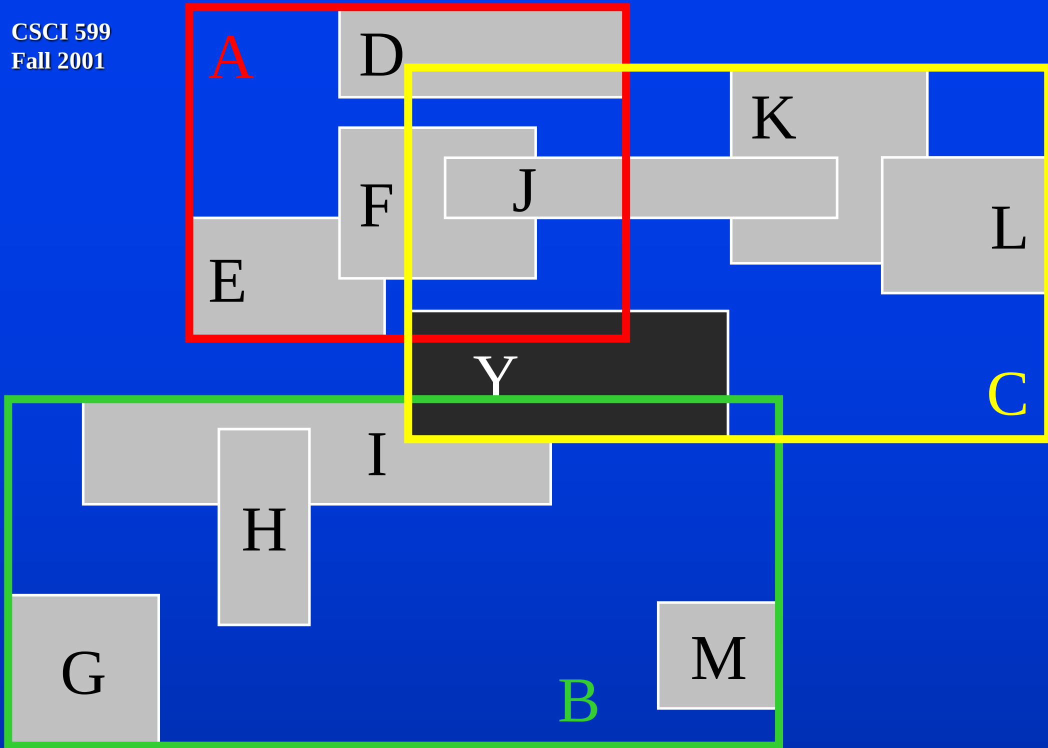
ChooseSubtree



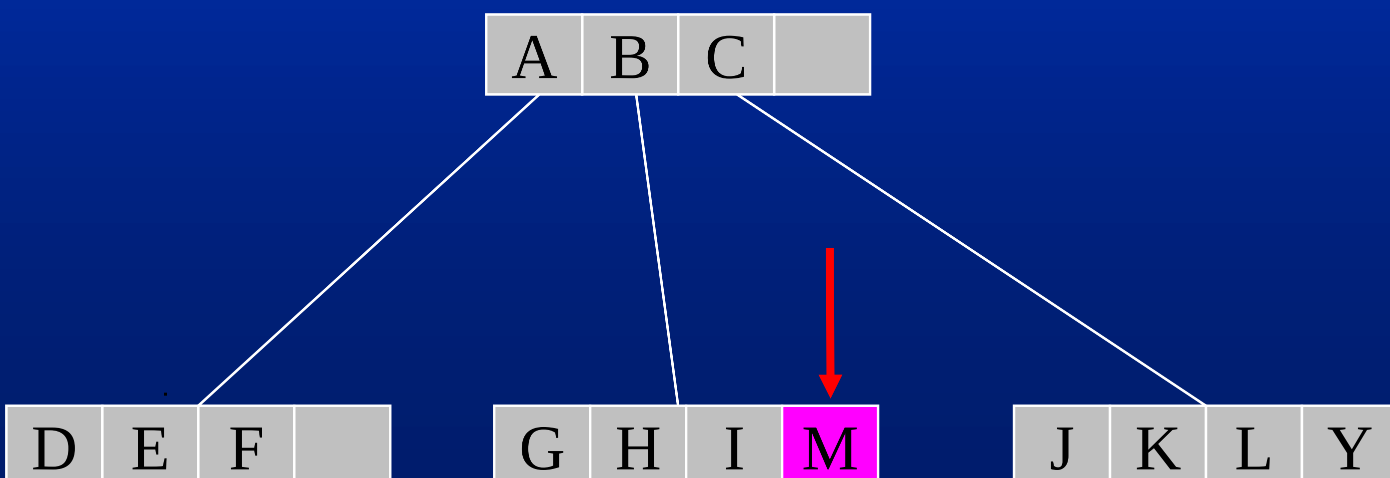
ReInsert

Insert

ChooseSubtree



Split is prevented!



Forced Reinsert

- p = 30% of M for all nodes yields best performance
- Change entries between neighboring nodes
  - decrease overlap
- Improve storage utilization
- Due to more restructuring, less splits occur
- Outer rectangles of a node are reinserted
  - shape of directory rectangles more quadratic
- Higher CPU cost (more insertion calls), alleviated by less splits

Experimental Setup

- Four R-tree variants
  - R-tree with quadratic split algorithm (qua.Gut)
  - R-tree with linear split algorithm (lin.Gut)
  - Greene’s variant of R-tree (Greene)
  - R*-tree

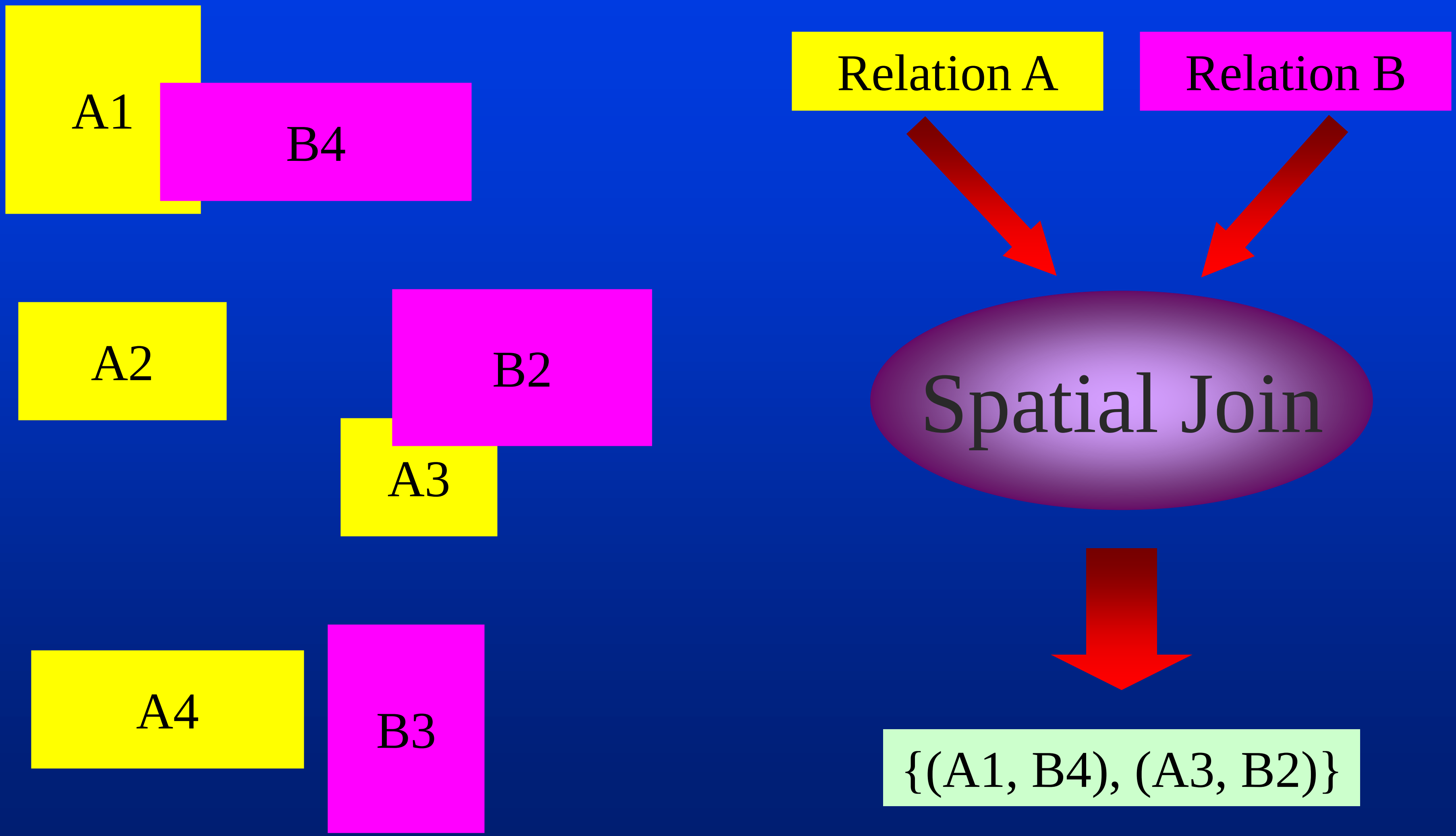
Experimental Setup

- Six data files containing about 100,000 2D rectangles
- Distribution of centers of rectangles
  - Uniform: 2D independent uniform distribution
  - Cluster: distribution with 640 clusters,1600 object/cluster
  - Parcel: 100,000 disjoint rectangles. Expand each area by factor of 2.5
  - Real-Data: MBRs of elevation lines from real cartography data
  - Gaussian: 2D independent Gaussian distribution
  - Mixed-Uniform: 2D independent uniform distribution (add 1,000 large rectangles to 99,000 small rectangles, then merge)

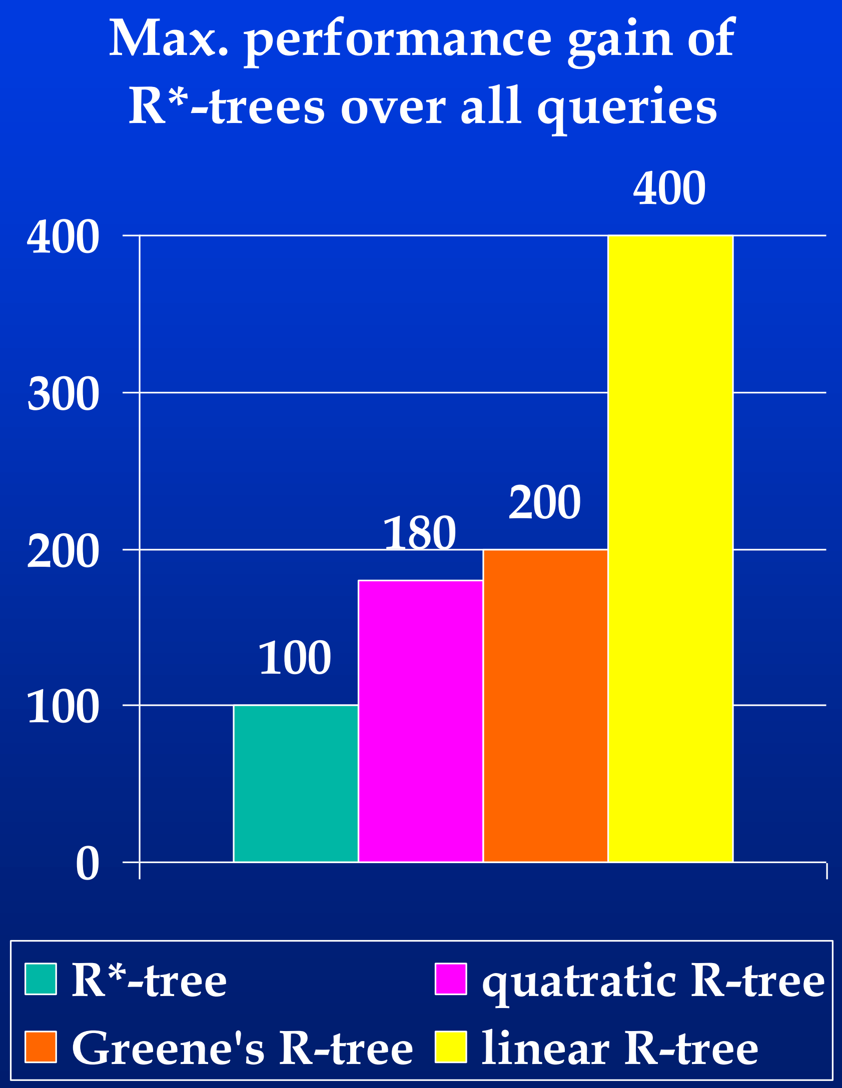
Experimental Setup

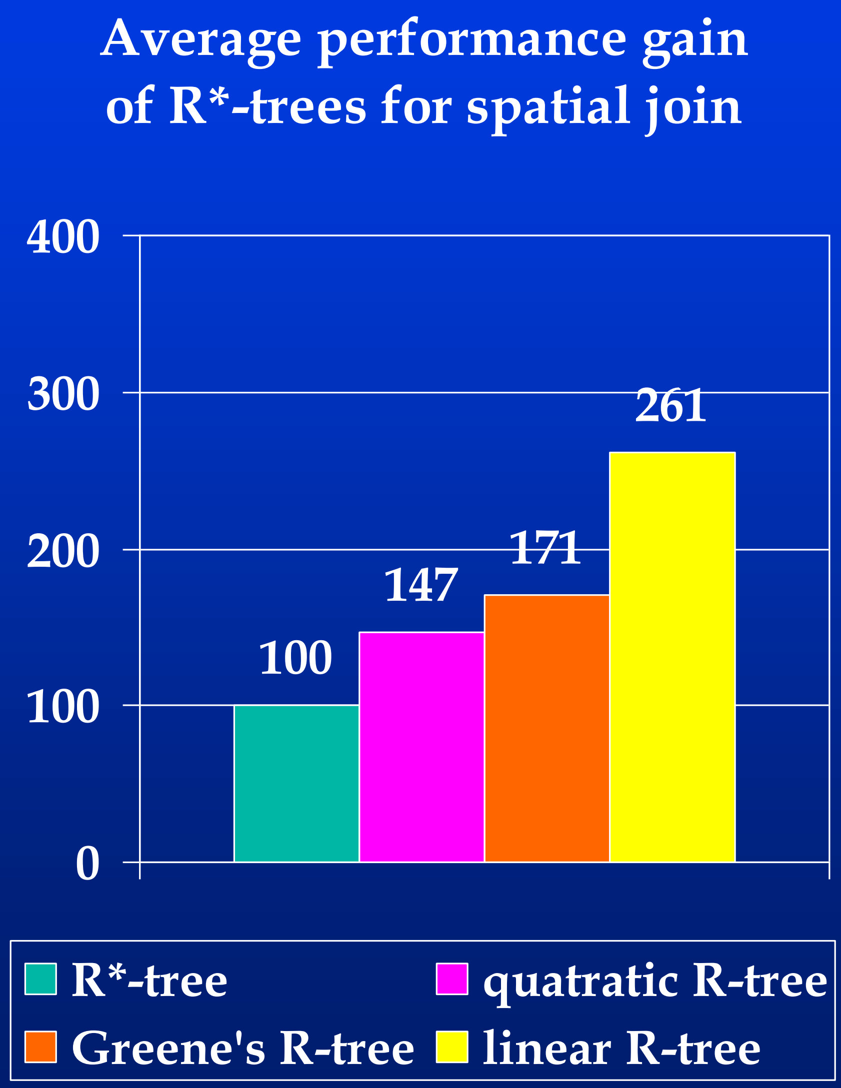
- Types of queries
  - Rectangle intersection query
    - given a rectangle S, find all rectangles R in the file with $R \cap S \neq \emptyset$
  - Point query
    - given a point P, find all rectangles R in the file with $P \in R$.
  - Rectangle enclosure query
    - given a rectangle S, find all rectangles R in the file with $R \supseteq S$
  - Spatial join
- All experiments measured in number of disk access

Spatial Join (Map Overlay)



Results of Experiments





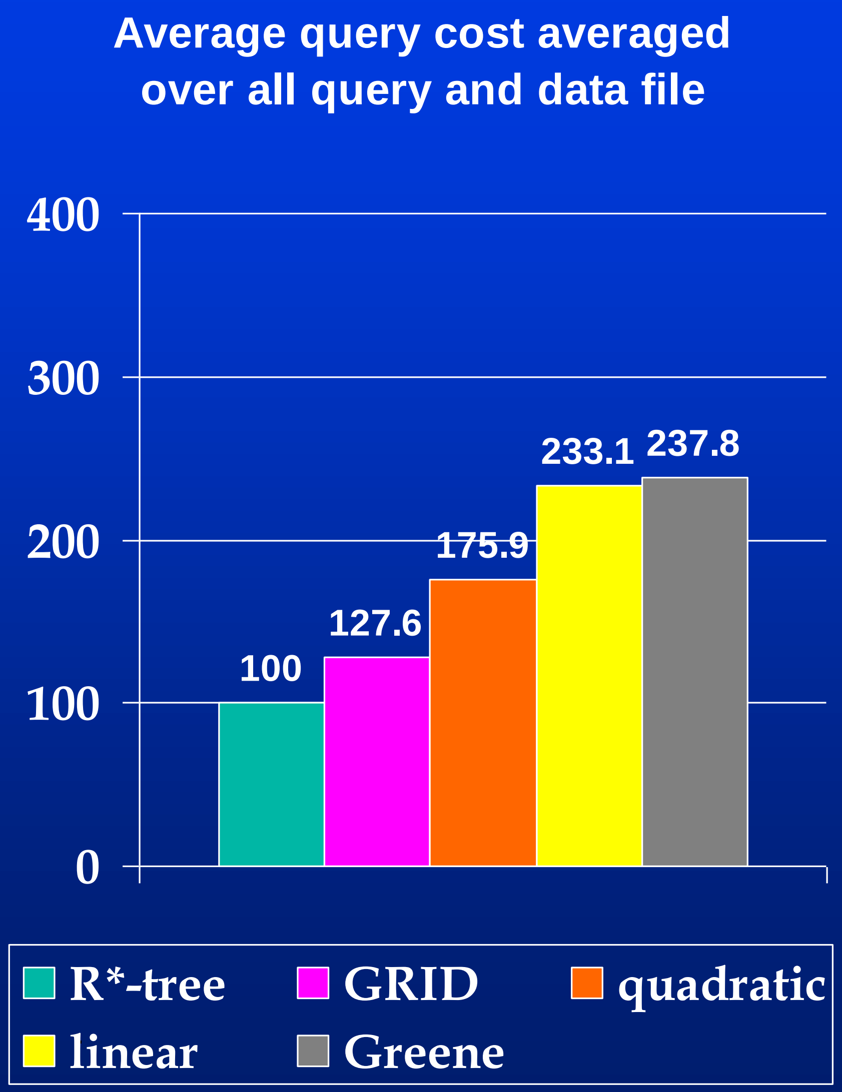
Results of Experiments

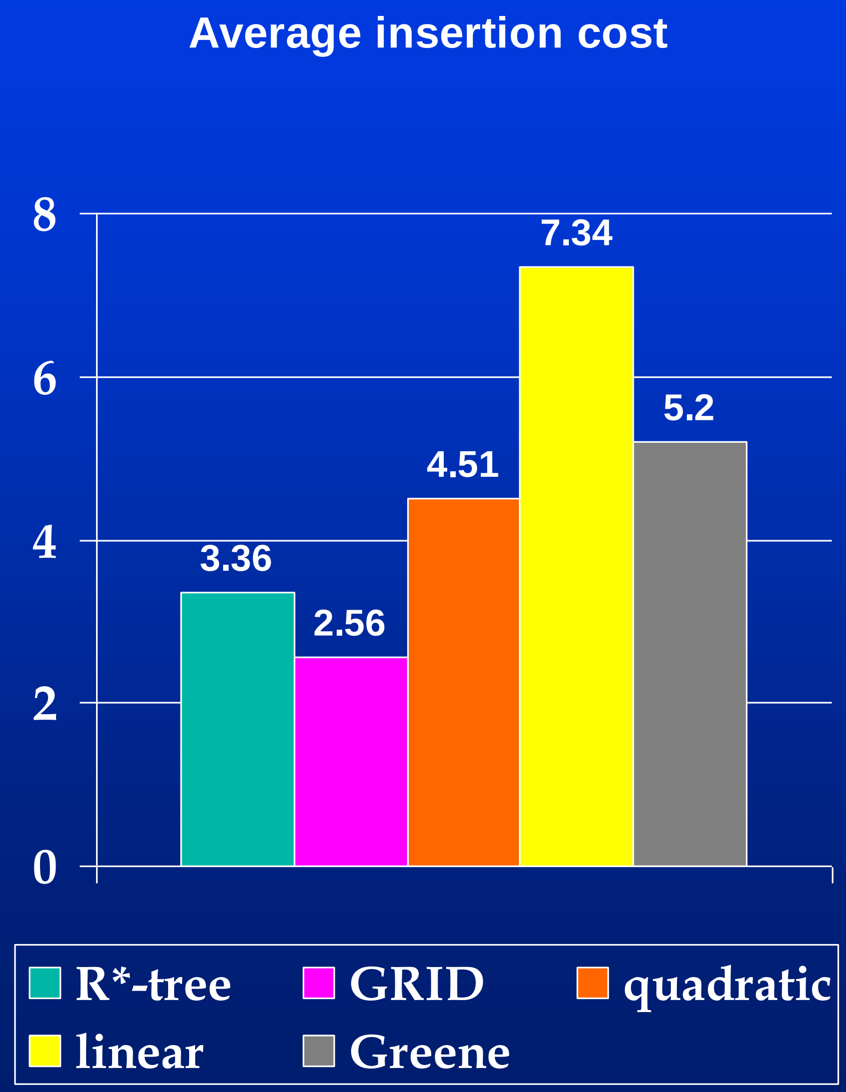
- R*-tree outperforms R-tree variants in all experiments
- Higher gain in R*-tree for smaller query rectangles
  - storage utilization gets more important for larger rectangles
- R*-tree has best storage utilization
- Despite use of Forced Reinsert, average insertion cost of R*-tree is still lowest

R*-tree: an Efficient Point Access Method

- Another experiment on benchmark for PAM [KSSS 89]
  - include 2-level grid file, a very popular PAM
- Performance gain considerably higher for points than for rectangles
- R*-tree outperforms R-tree variants for point data and storage utilization, even grid file

Results of Experiments





Conclusion

- R*-tree can efficiently be used as an access method in database systems organizing both multidimensional points and spatial data
- Since area, margin and overlap are reduced, R*-tree is very robust against ugly data distribution