EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

g	6	6	t

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

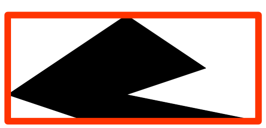
Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

R-tree

\

- In multidimensional space, there is no unique ordering! Not possible to use B+-trees☹
- [Guttman 84] R-tree!
- Group objects close in space in the same node
  => guaranteed page utilization
  => easy insertion/split algorithms.
  - (only deal with Minimum Bounding Rectangles - **MBR**s)



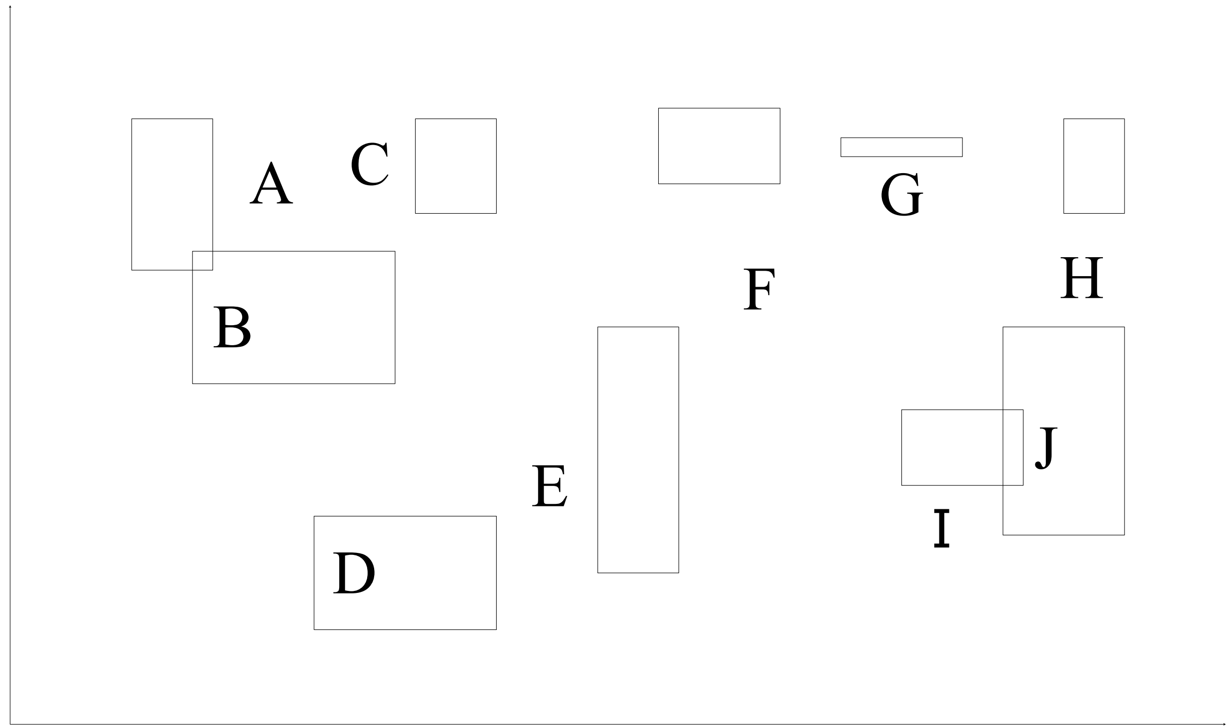
R-tree

\

- A multi-way external memory tree
- Keys: n-dimensional rectangles, (2 points)
- Index nodes and data (leaf) nodes
- All leaf nodes appear on the same level
  - Leaf node index entries: (I, tuple_id)
  - Non-leaf node entry: (I, child_ptr)
- Every node contains between m and M entries
  - $m \leq M/2$ is the minimum entries per node.
- The root node has at least 2 entries (children)

Example

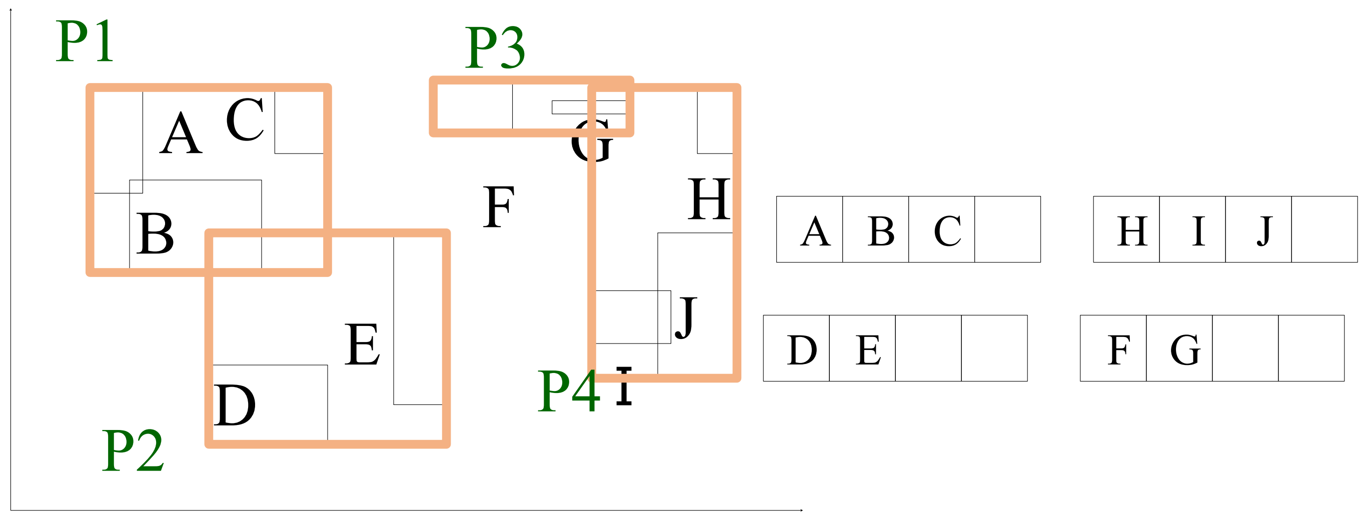
eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page



Example

- R trees grow like B+ trees
  - Bottom up

    eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page

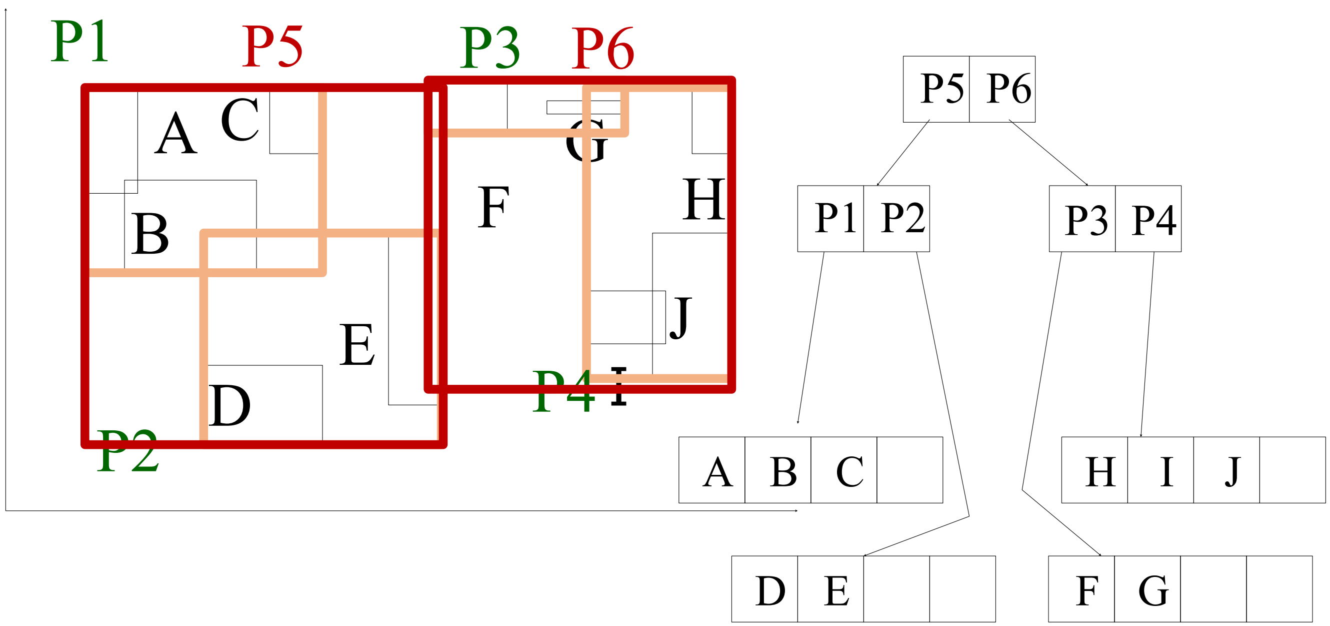


Example

\

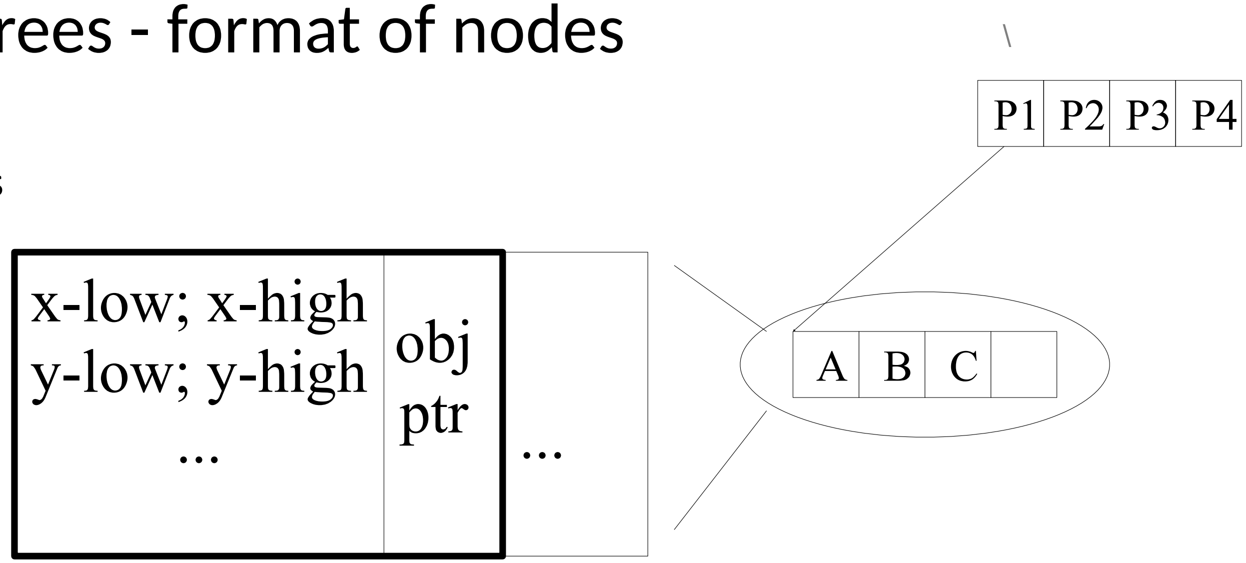
- R trees grow like B+ trees
  - Bottom up

    eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page

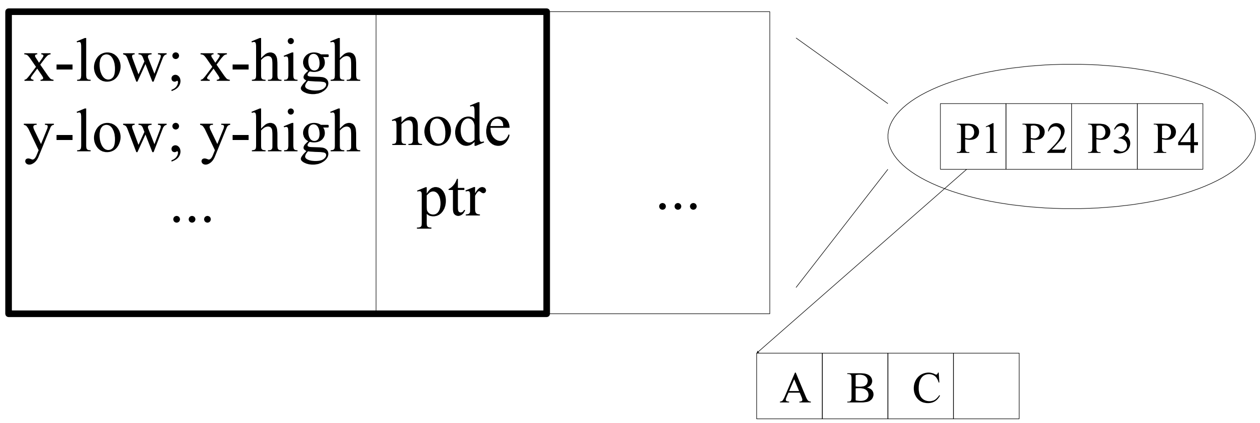


R-trees - format of nodes

- {(MBR; obj_ptr)} for leaf nodes



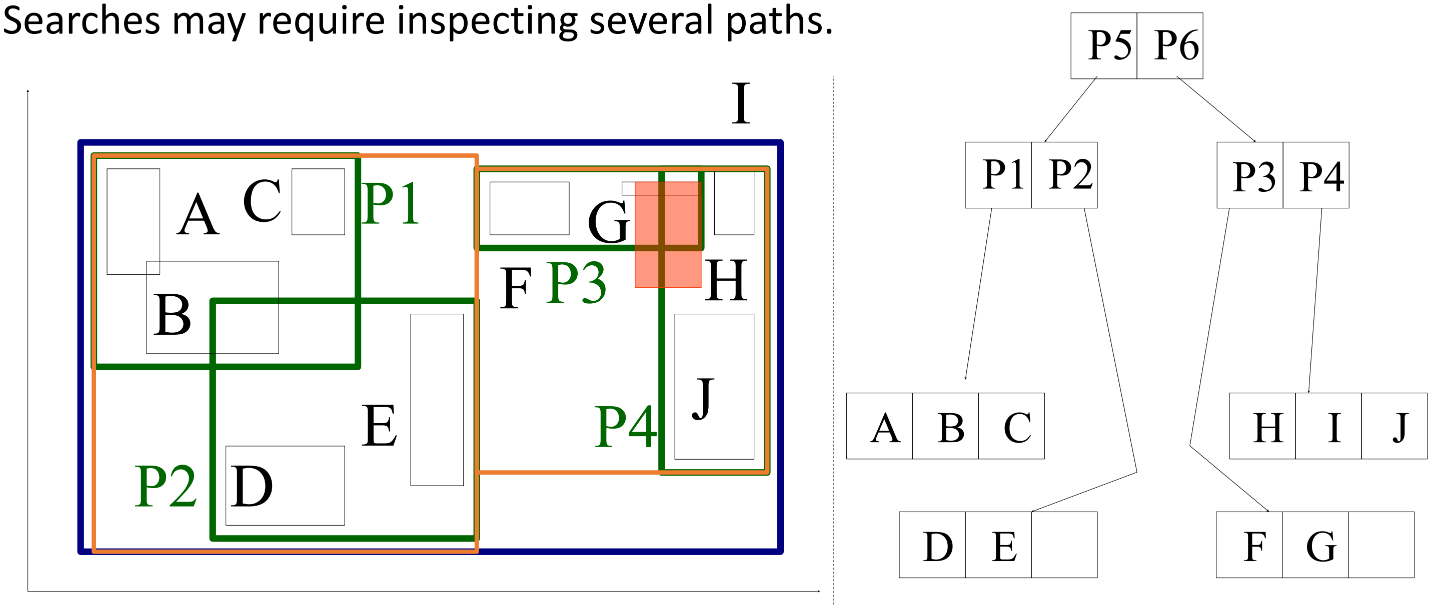
- {(MBR; node_ptr)} for non-leaf nodes



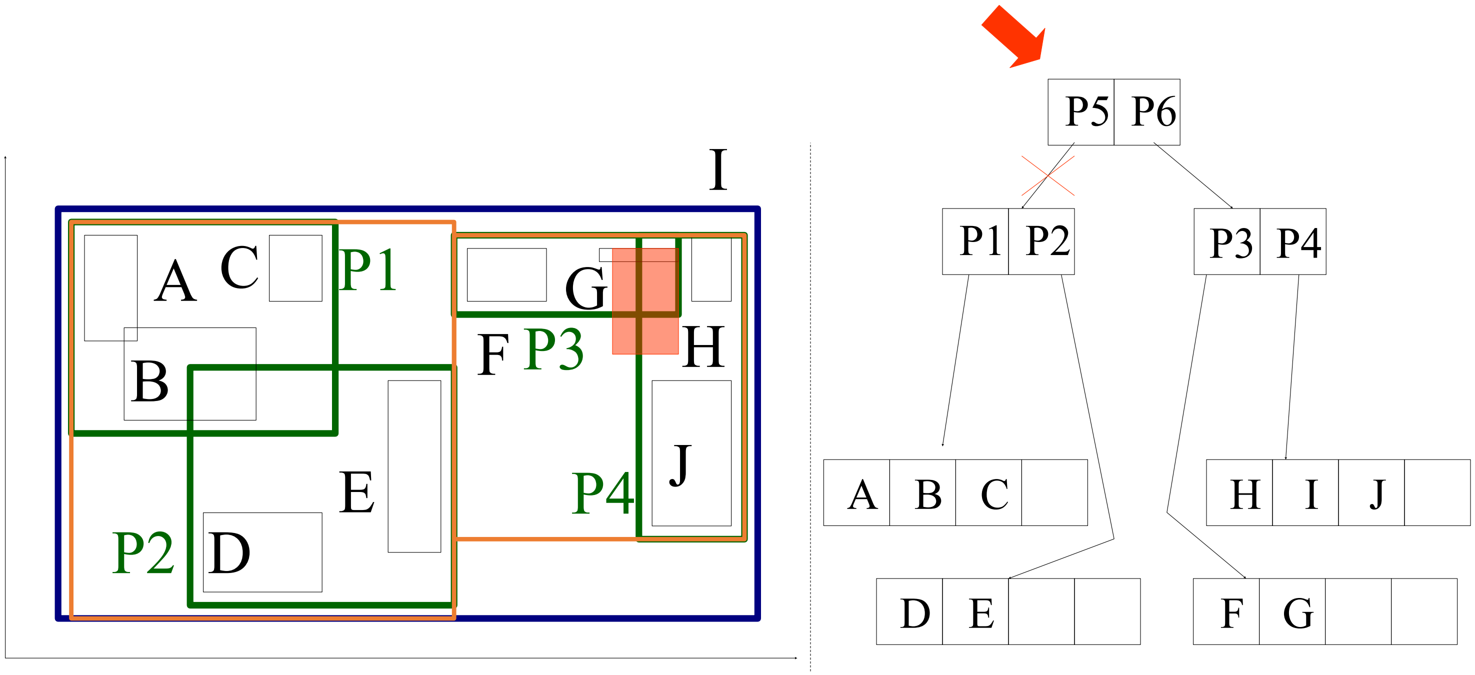
R-trees: Search

\

- Given a search rectangle S ...
  - Start at root and locate all child nodes whose rectangle I intersects S (via linear search).
  - Search the subtrees of those child nodes.
  - When you get to the leaves, return entries whose rectangles intersect S.
  - Searches may require inspecting several paths.
-



R-trees:Search



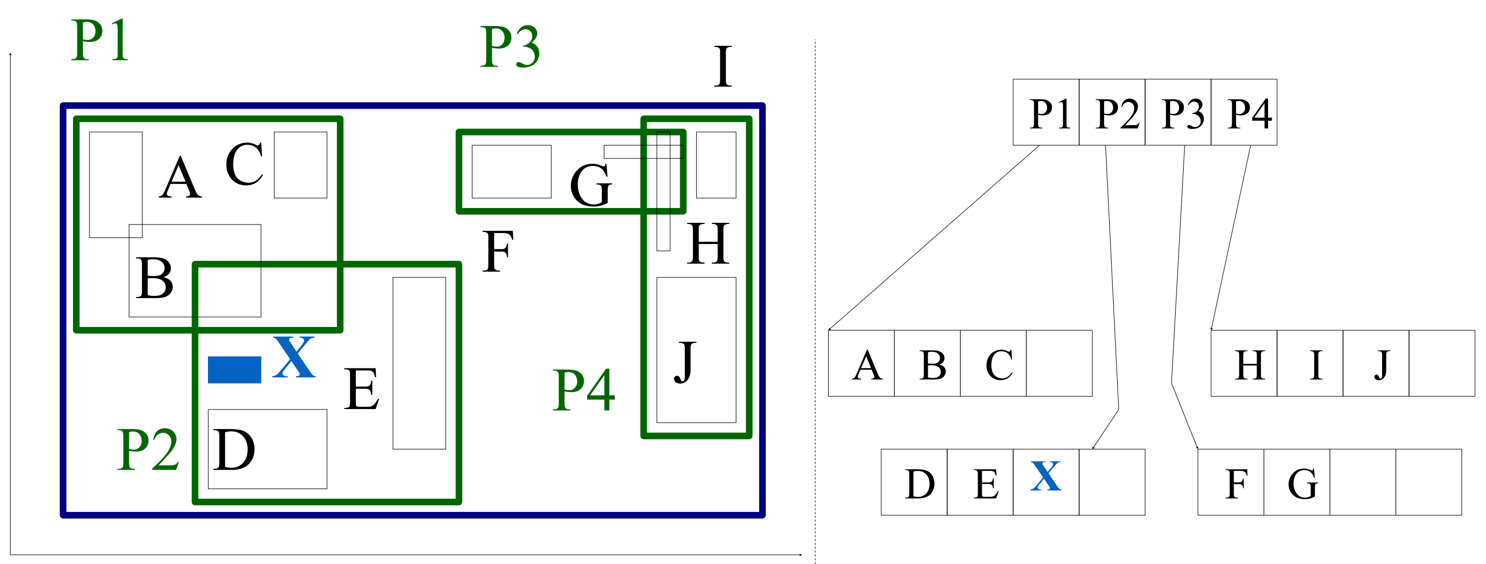
R-trees: Search

- Main points:
  - every parent node completely covers its 'children'
  - nodes in the same level may overlap!
  - a child MBR may be covered by more than one parent - it is stored under ONLY ONE of them
  - a point query may follow multiple branches.
  - works for higher dimensions

R-trees:Insertion

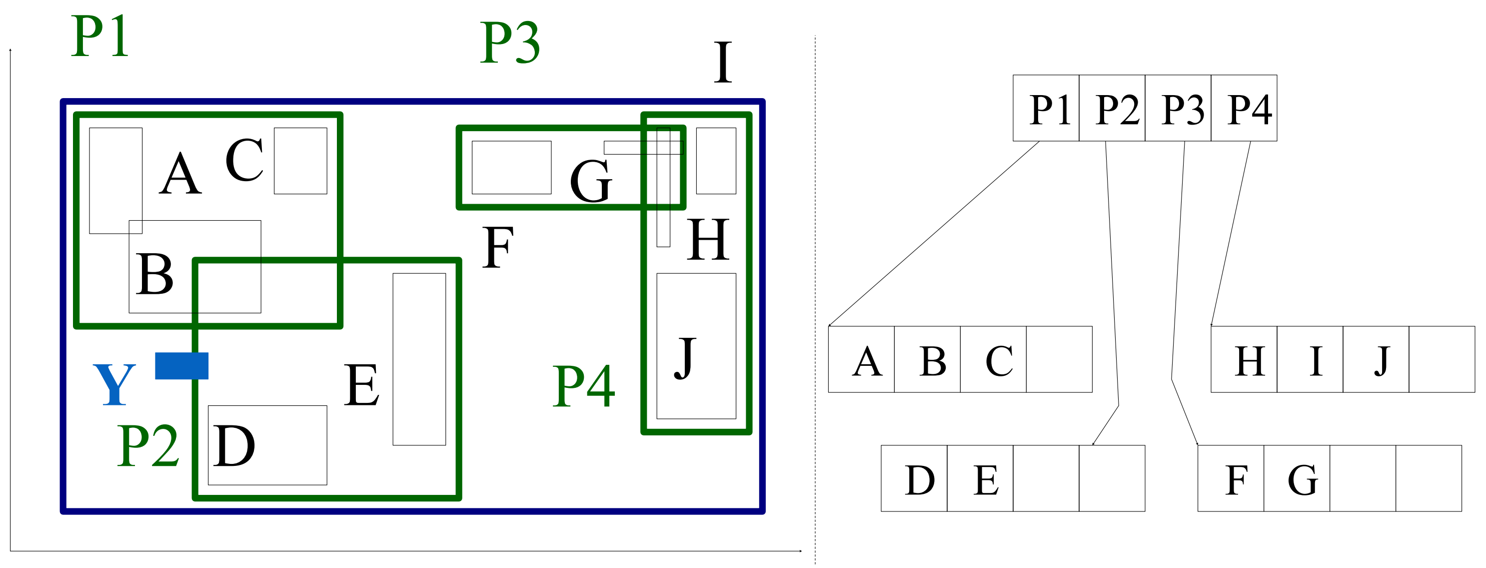
\

- Insert X: Start from the leaves. Which one?
  - Start at root
  - Go down the tree by choosing child whose rectangle needs the least enlargement to include X (Δ area or perimeter…) In case of a tie, choose child with smallest area
  - Least enlargement: increase in area or perimeter…a choice!



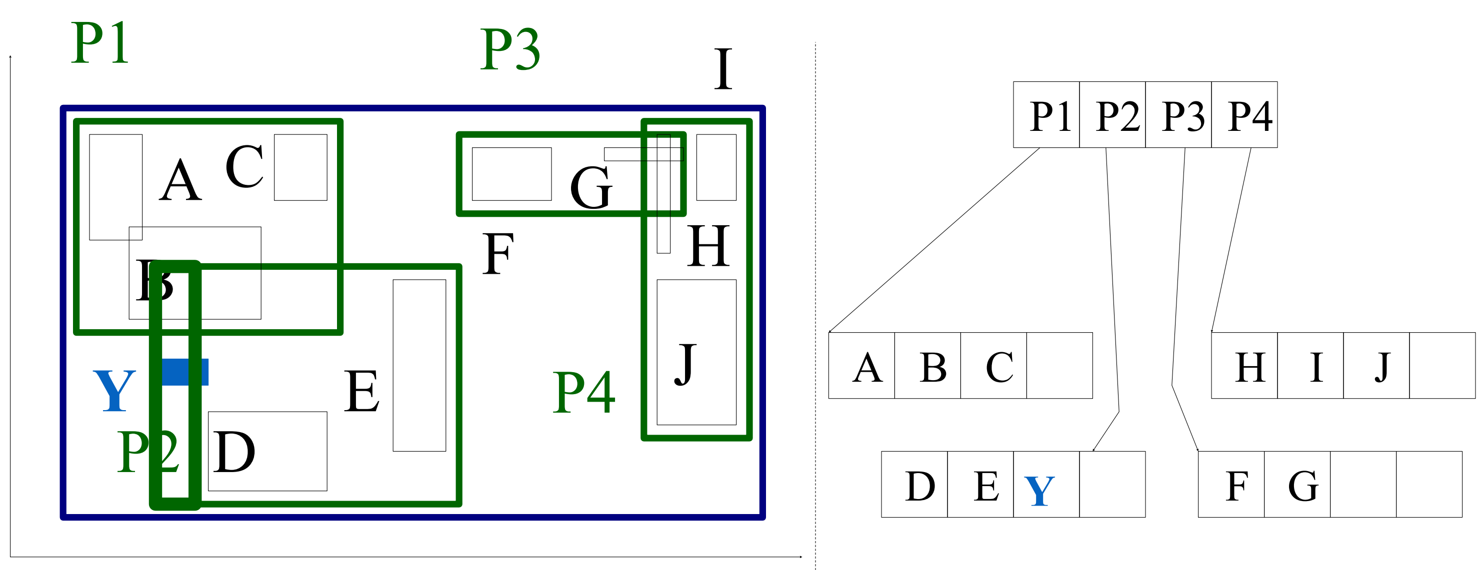
R-trees:Insertion

Insert Y



R-trees:Insertion

- Extend the parent MBR



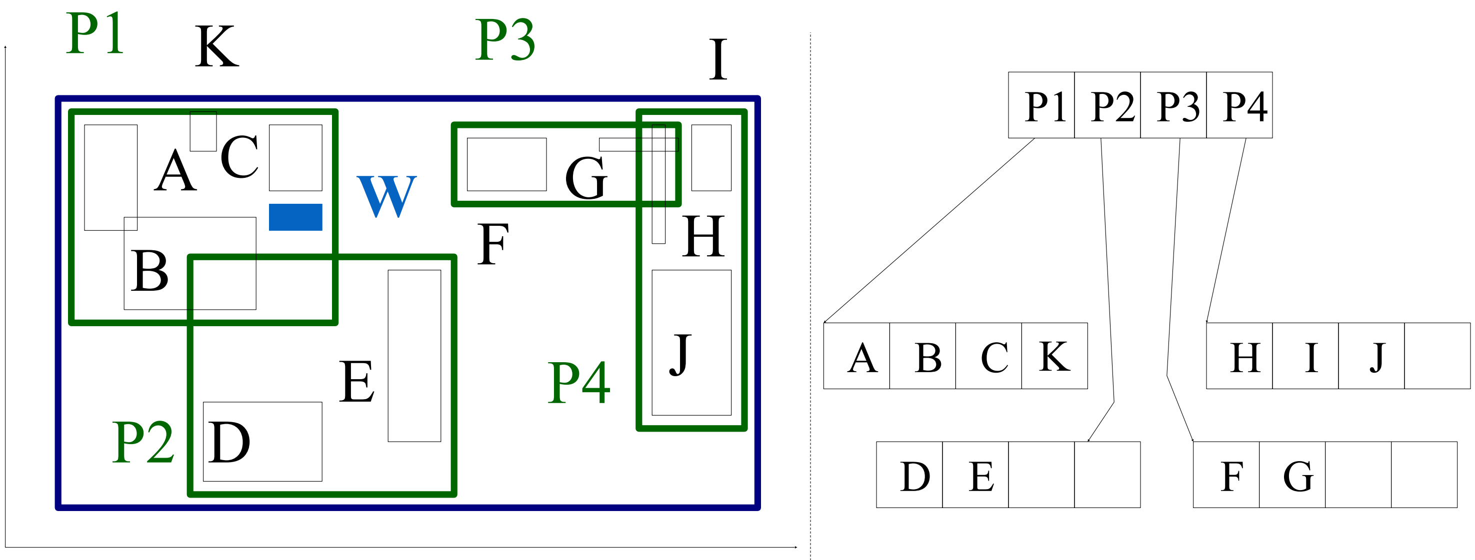
R-trees:Insertion

\

- How to find the next node to insert a new object Y?
  - Using ChooseLeaf: Find the entry that needs the least enlargement to include Y. Resolve ties using the area (smallest)
- Enlargement measured by change in perimeter of MBR or change in area
- Problem: Can saturate a leaf. In this case, need to ***split***
  - When you split, you readjust MBR in parent to correspond to remaining objects in each of the new nodes.
  - May need to recursively split parent…

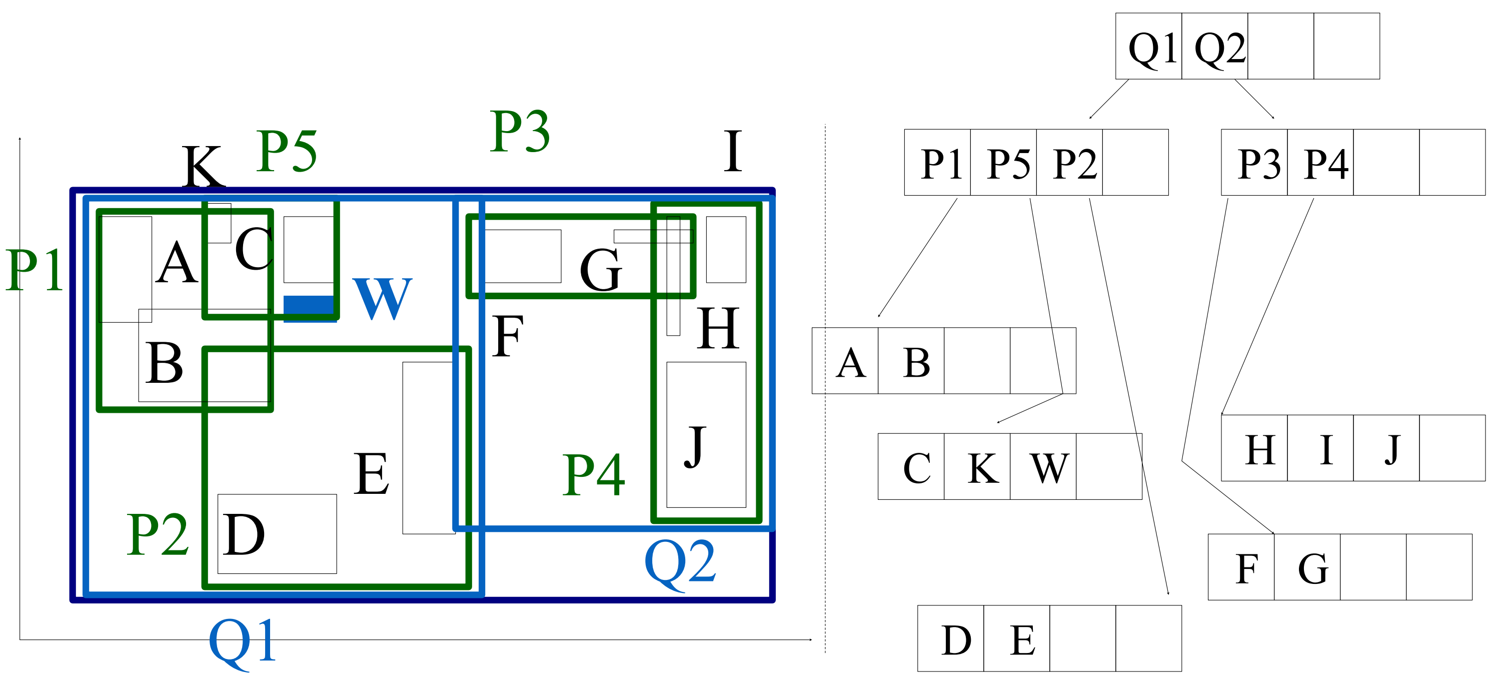
R-trees:Insertion

- If node is full then <u>Split :</u> ex. Insert w



R-trees:Insertion

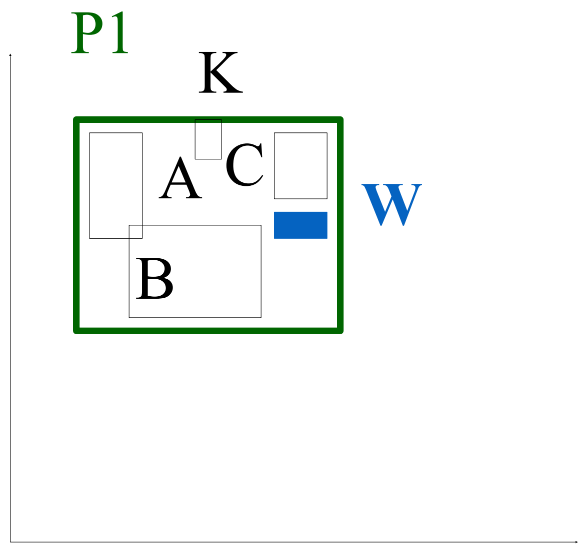
- If node is full then <u>Split :</u> ex. Insert w
- Note shrinkage of P_1



R-trees:Split

\

- Split node P1: partition the MBRs into two groups.
- Multiple algorithms possible



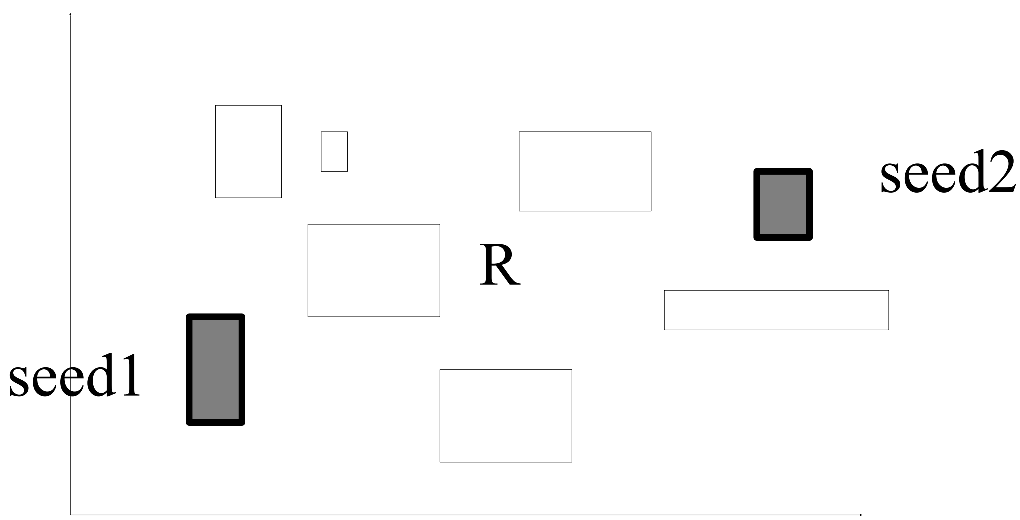
- A2: 'linear' split
- A3: quadratic split
- A4: exponential split:

  2^{M-1} choices

R-trees:Split

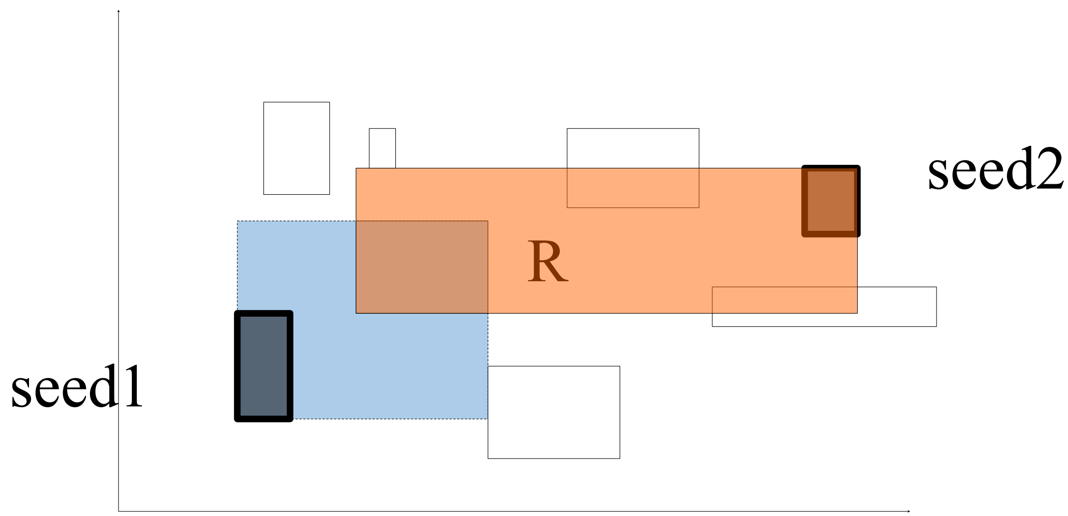
\

- Pick two rectangles as ‘seeds’for group 1 and group 2
  - Farthest apart in dimension relative to total spread in dimension



R-trees: Split

- pick two rectangles as ‘seeds’ for group 1 and group 2;
- assign each rectangle ‘R’ to the ‘closest’ ‘group’ in any order
- ‘closest’: the smallest increase in area
- Once a base rectangle has maximum number of rectangles for split, the rest are assigned to other rectangle: guarantee minimum m in both!



R-trees: Linear Split

\

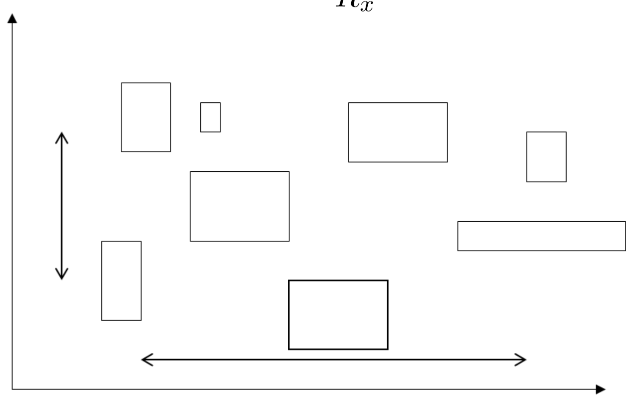
- How to pick Seeds:
  - Find the rects with the highest low and lowest high sides in each dimension
  - Normalize the separations by dividing by the width of all the rects in the corresponding dim
  - Choose the pair with the greatest normalized separation

$$\text{Rectangles } [(x_{low}^{(i)}, y_{low}^{(i)}), (x_{hi}^{(i)}, y_{hi}^{(i)})]$$

$$y_{hi} = \max_i y_{low}^{(i)}; \quad y_{low} = \min_i y_{hi}^{(i)}; \quad x_{low} = \min_i x_{hi}^{(i)}; \quad x_{hi} = \max_i x_{low}^{(i)}$$

$$R_y = \max_i y_{hi}^{(i)} - \min_i y_{low}^{(i)}; R_x = \max_i x_{hi}^{(i)} - \min_i x_{low}^{(i)}$$

$$\text{Normalized Separation: } NS(x) = \frac{x_{hi} - x_{low}}{R_x}; \quad NS(y) = \frac{y_{hi} - y_{low}}{R_y}$$



R-trees: Quadratic Split

\

- How to pick Seeds:
  - For each pair E1 and E2, calculate the rectangle J=MBR(E1, E2) and d= J-E1-E2. Choose the pair with the largest d
- PickNext:
  - For each remaining rectangle E, calculate the area increase to include it in group d1(E) and d2(E)
  - Choose the remaining rectangle to insert with highest difference:

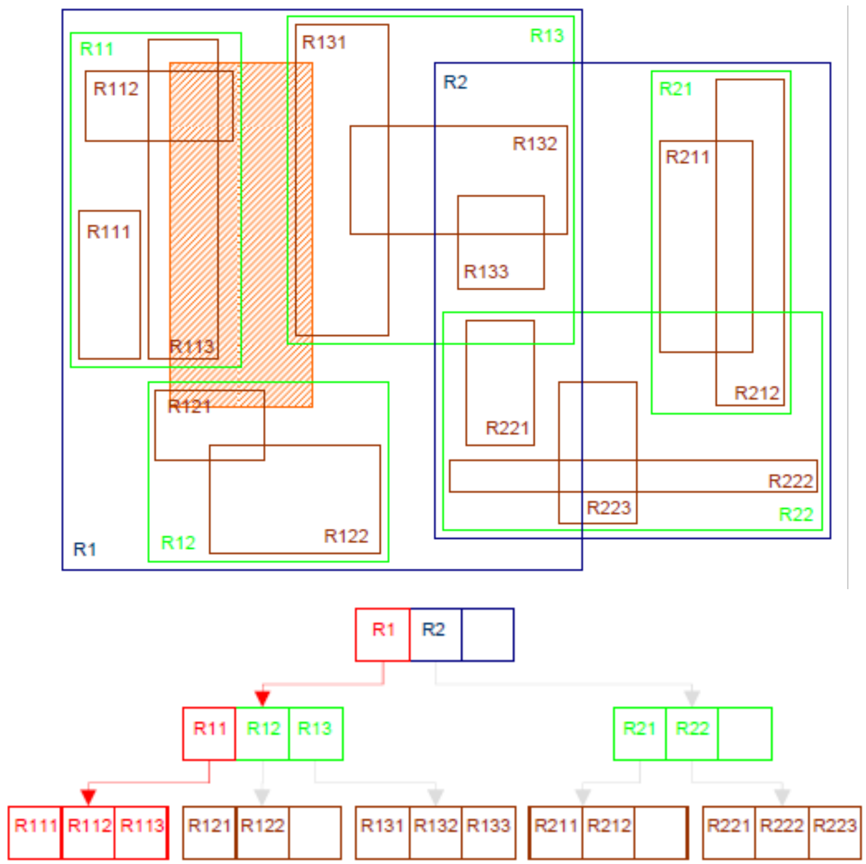
    |d1(E)-d2(E)|
  - Assign this remaining rectangle to its closest group: the one that has the smallest area increase.
  - Repeat until all rectangles are assigned, or until one group has M-m+1 entries. In the latter case, put the remaining rectangles into the other group and stop. If all rectangles have been distributed then stop.

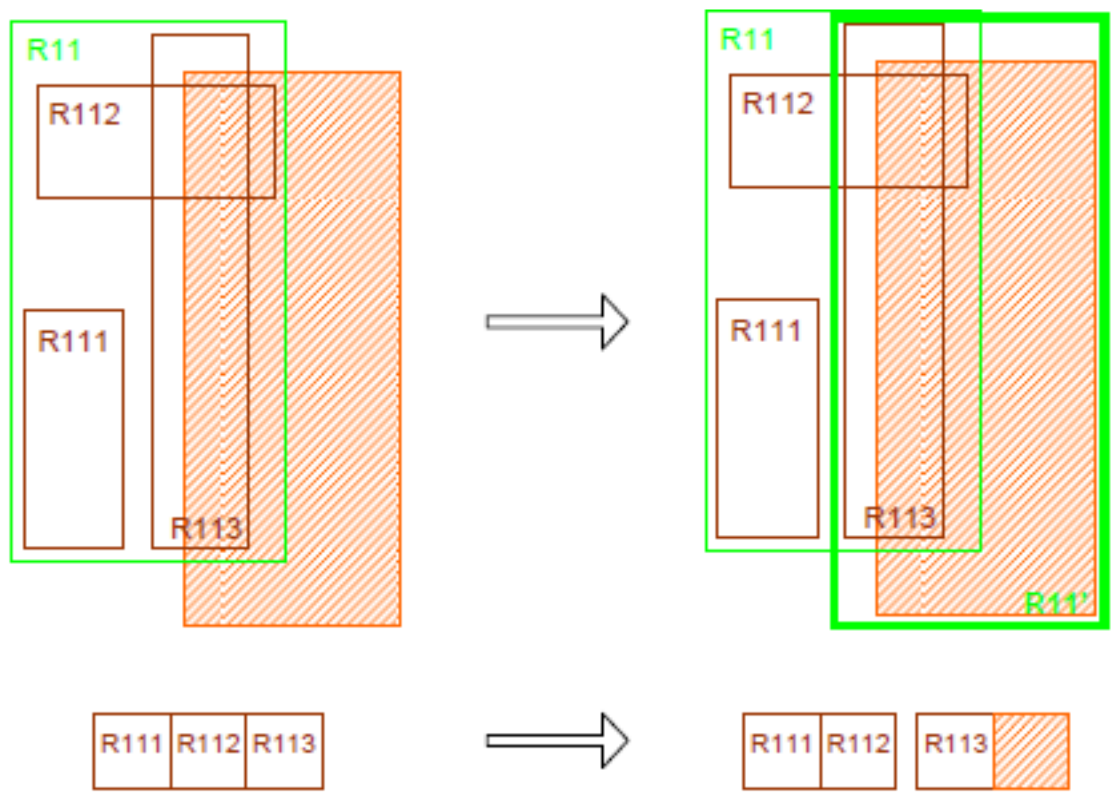
R-Trees:Deletion

\

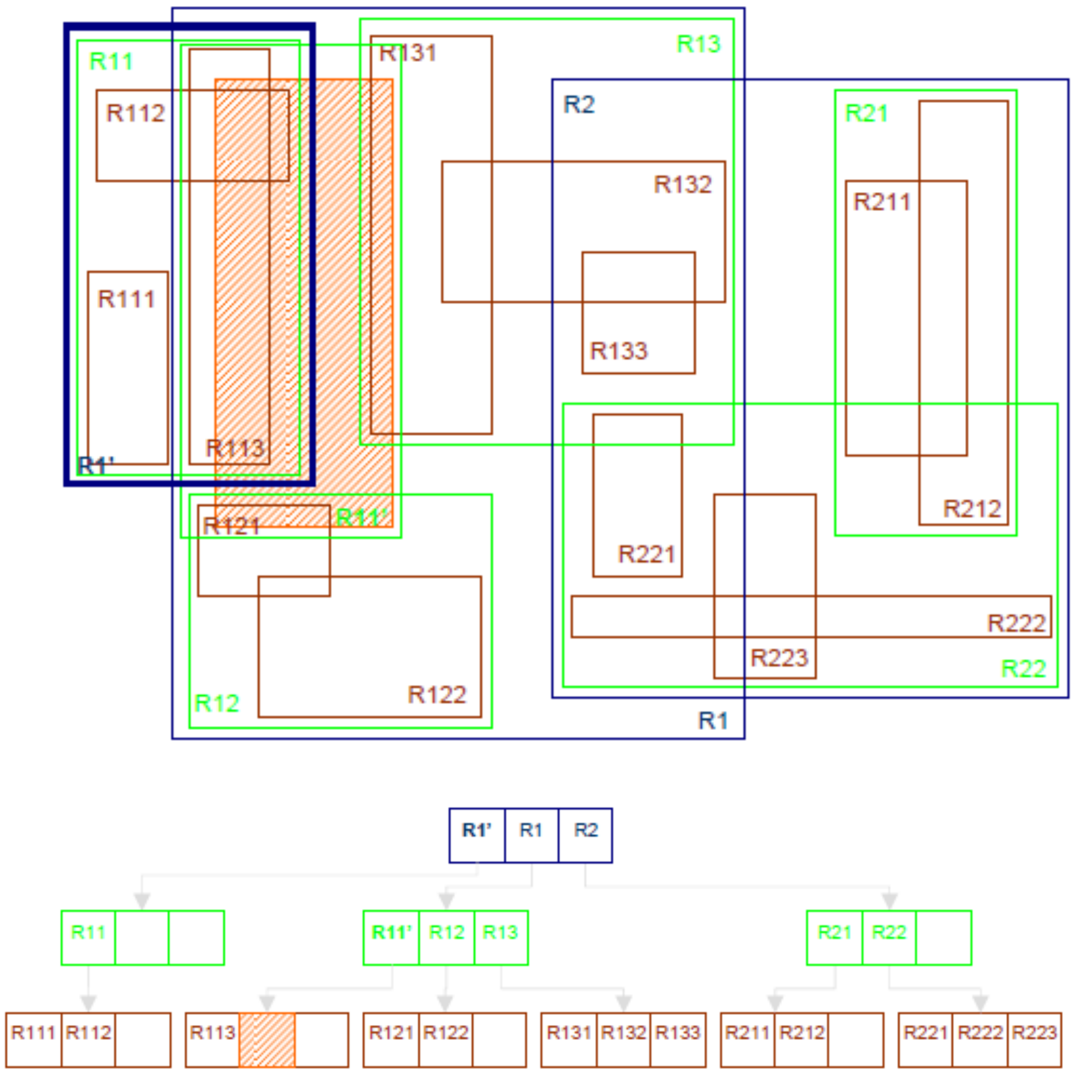
- Find the leaf node that contains the entry E
- Remove E from this node
- If underflow:
  - Eliminate the node by removing the node entries and the parent entry
  - Reinsert the orphaned (other entries) into the tree using **Insert**
- Why reinsert?
  - Nodes can be merged with sibling whose area will increase the least, or entries can be redistributed.
  - Reinsertion is easier to implement.

Insertion example





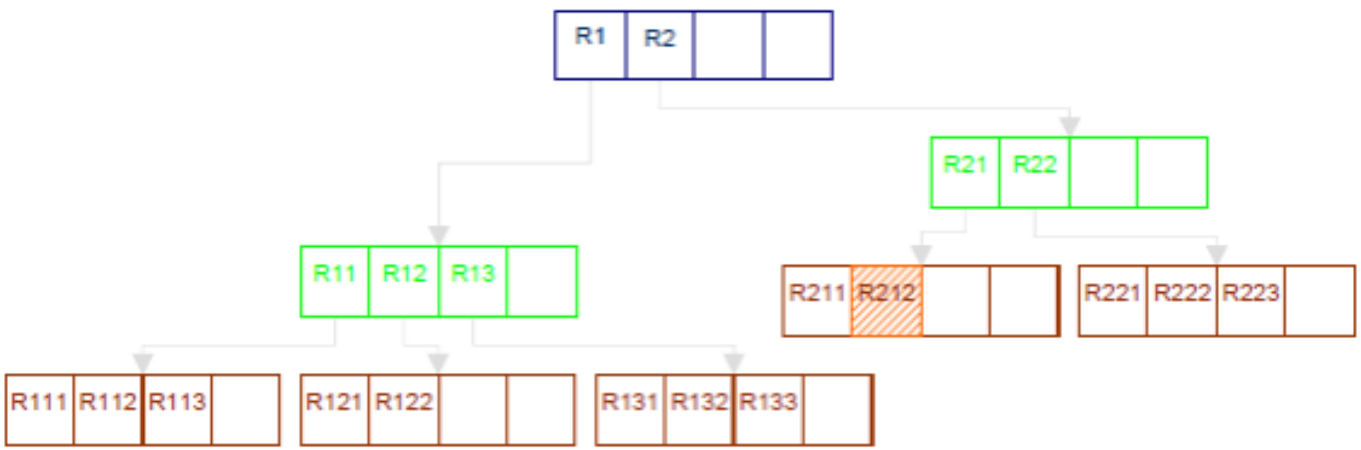
R11
R112
R111
R113
R11
R112
R111
R113
R11'
R111
R112
R113
R111
R112
R113



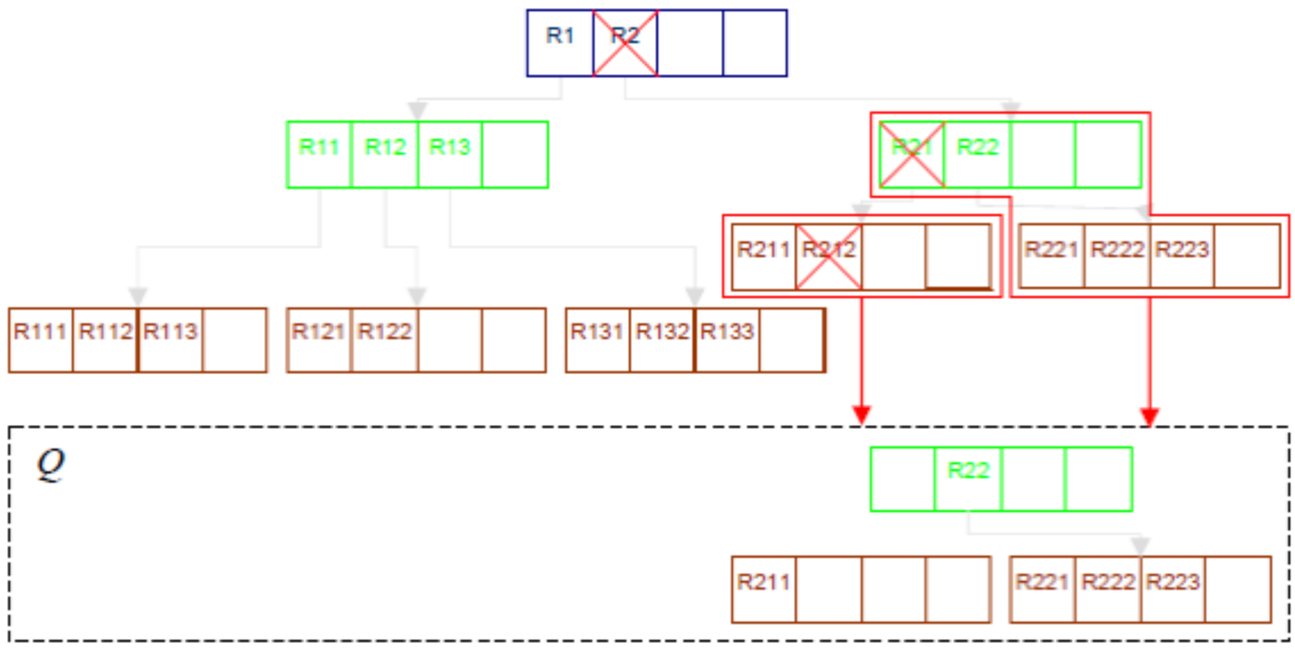
R11
R112
R111
R113
R1'
R131
R13
R2
R21
R132
R211
R133
R212
R121
R11'
R221
R222
R223
R22
R12
R122
R1
R1' R1 R2
R11
R11' R12 R13
R21 R22
R111 R112
R113
R121 R122
R131 R132 R133
R211 R212
R221 R222 R223

R-Trees: Deletion

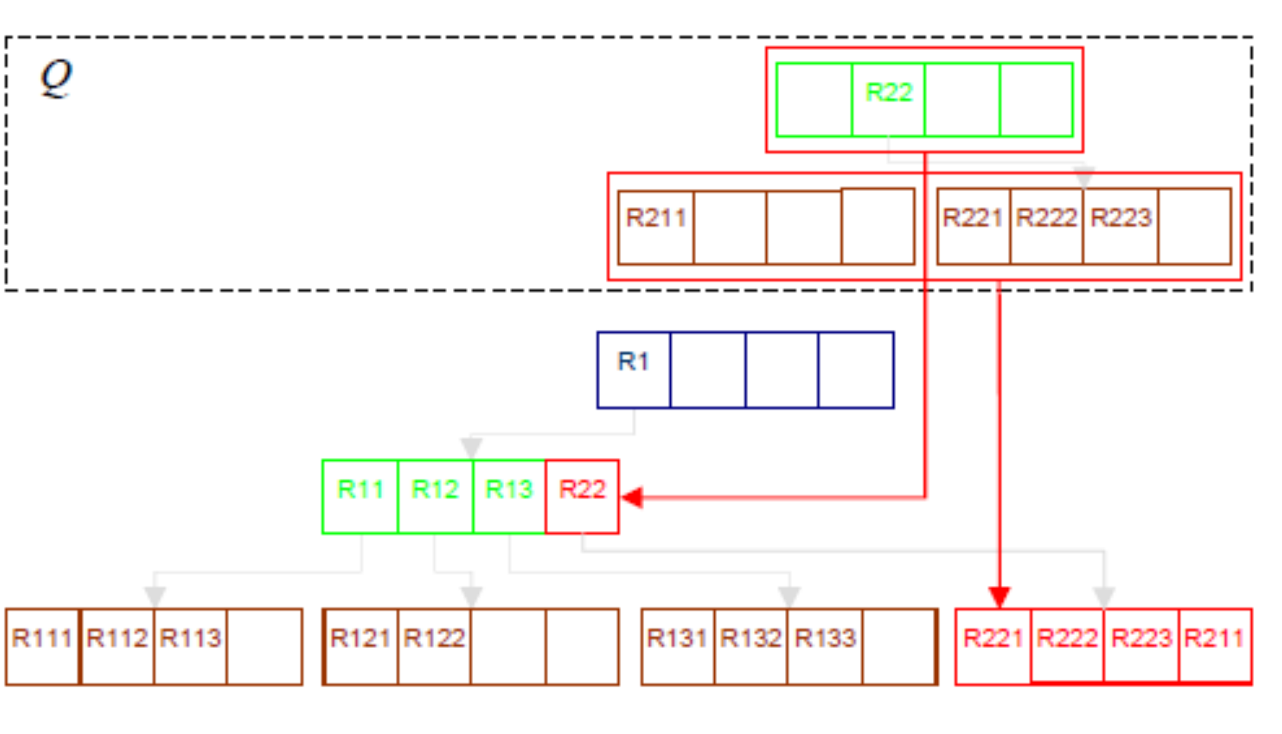
- M = 4, m = 2; Delete R212 which creates underflow in R21.
- Will delete and add R211 (orphan) to reinsert Queue
- Have underflow in R2. Will delete R2 and add intermediate node R2 to reinsert queue. Add the entries in R22 for reinsertion.



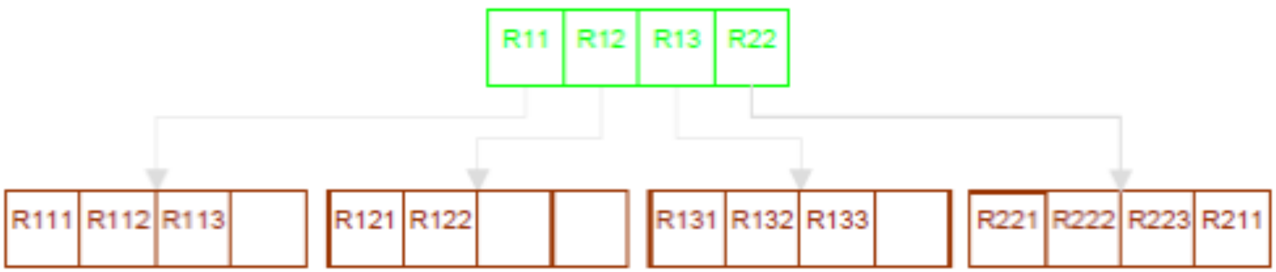
R-Trees: Deletion



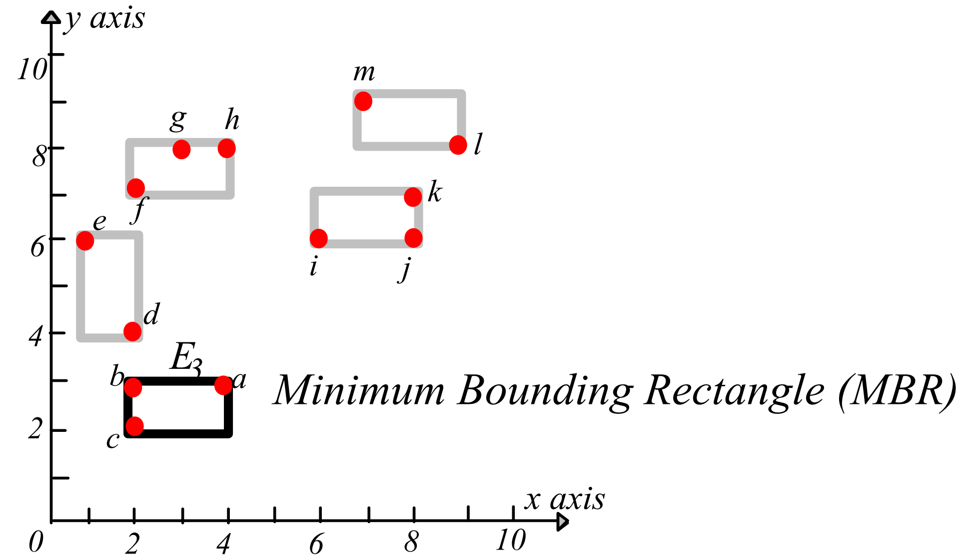
R-Trees: Deletion



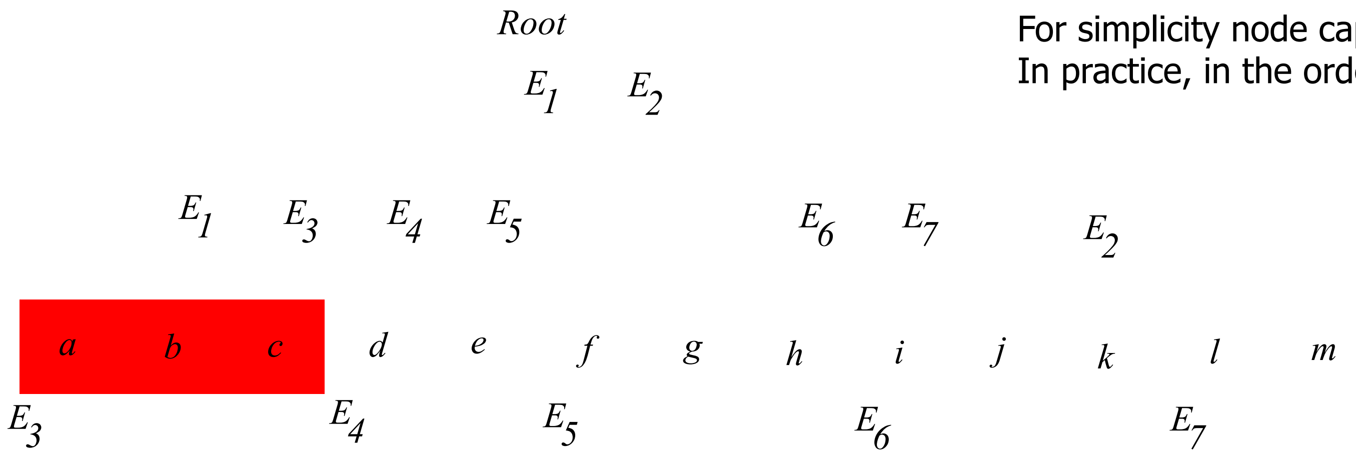
R-Trees: Deletion



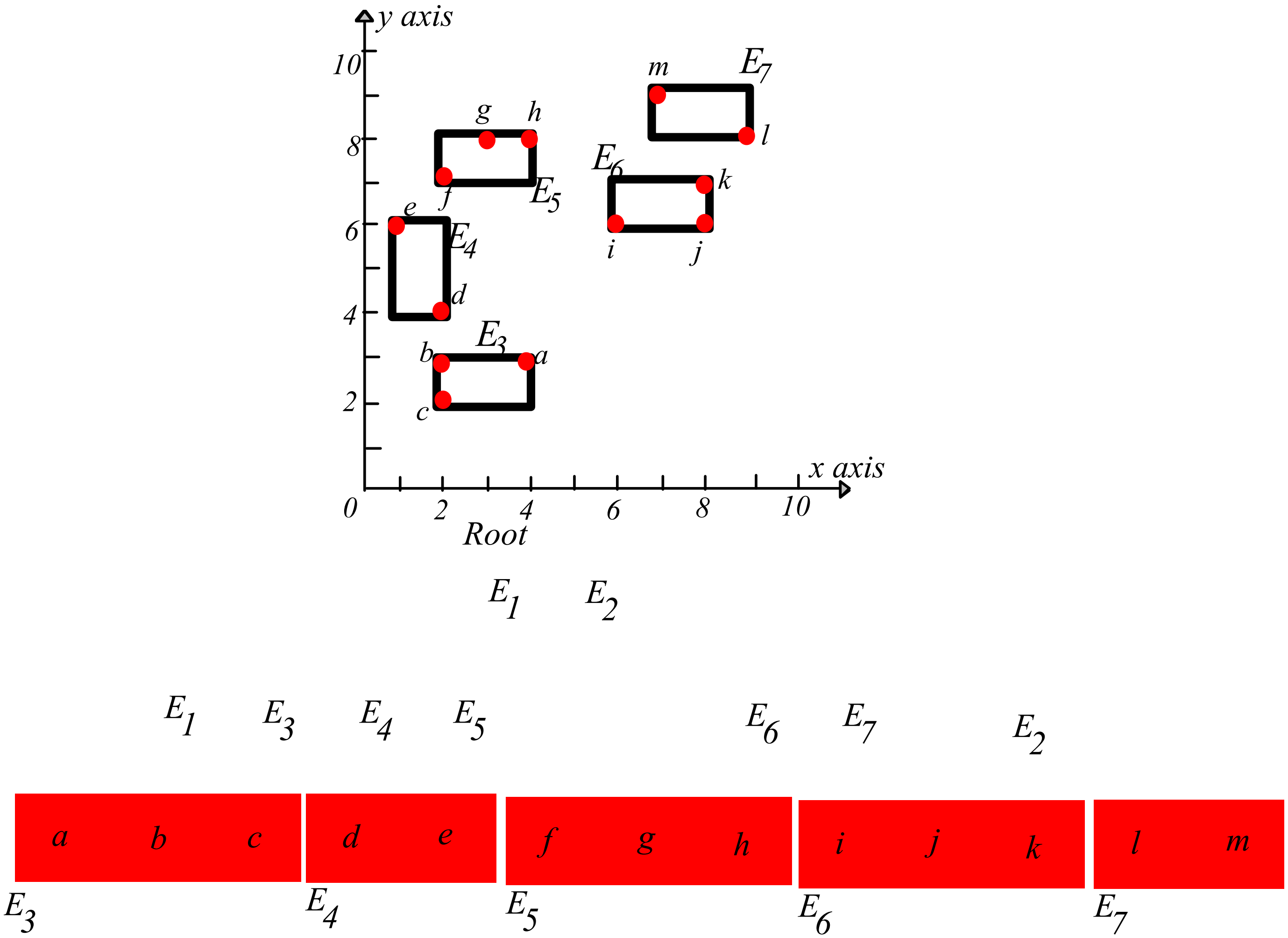
R-trees with point data



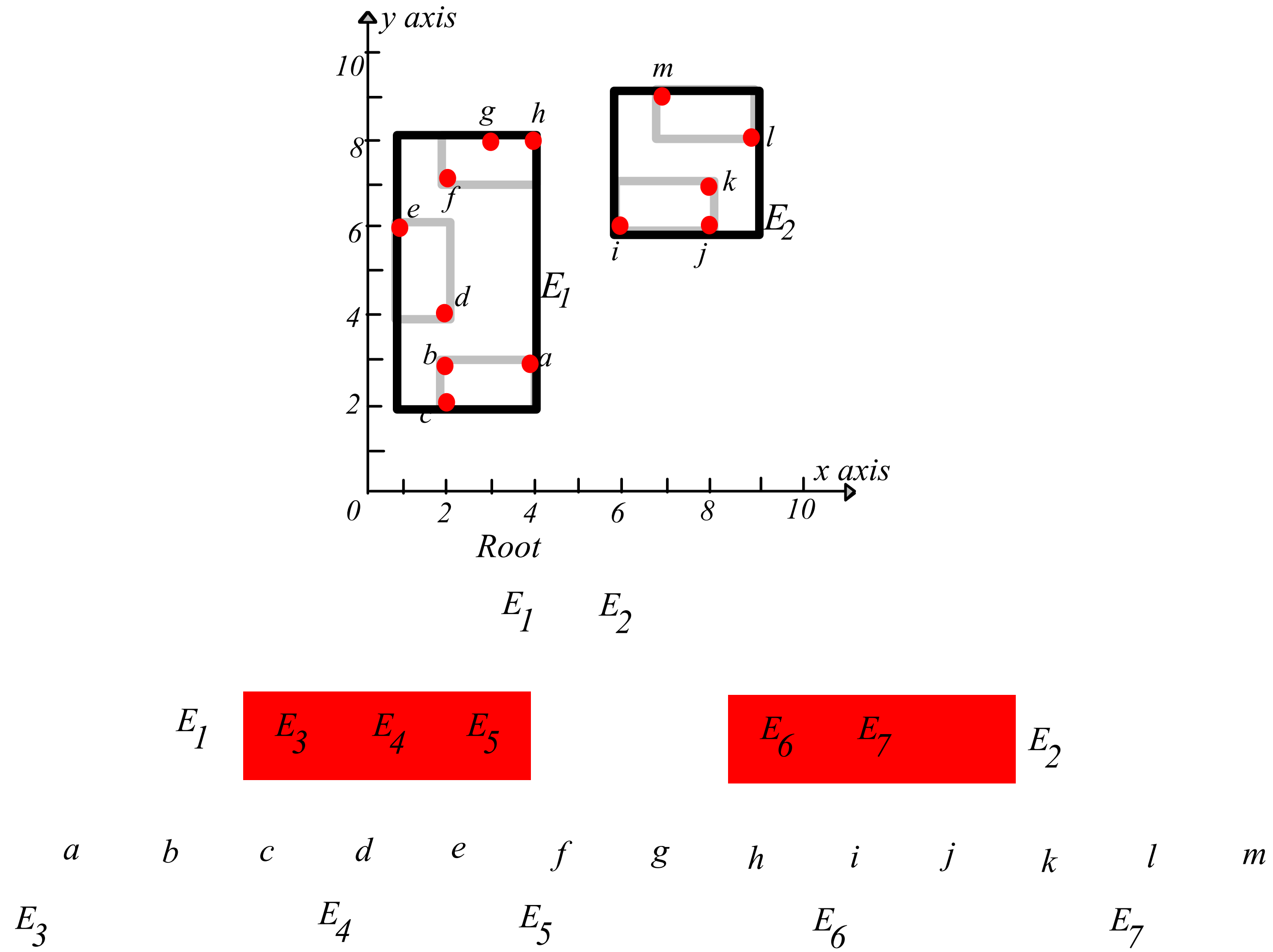
For simplicity node capacity = 3
In practice, in the order of 100



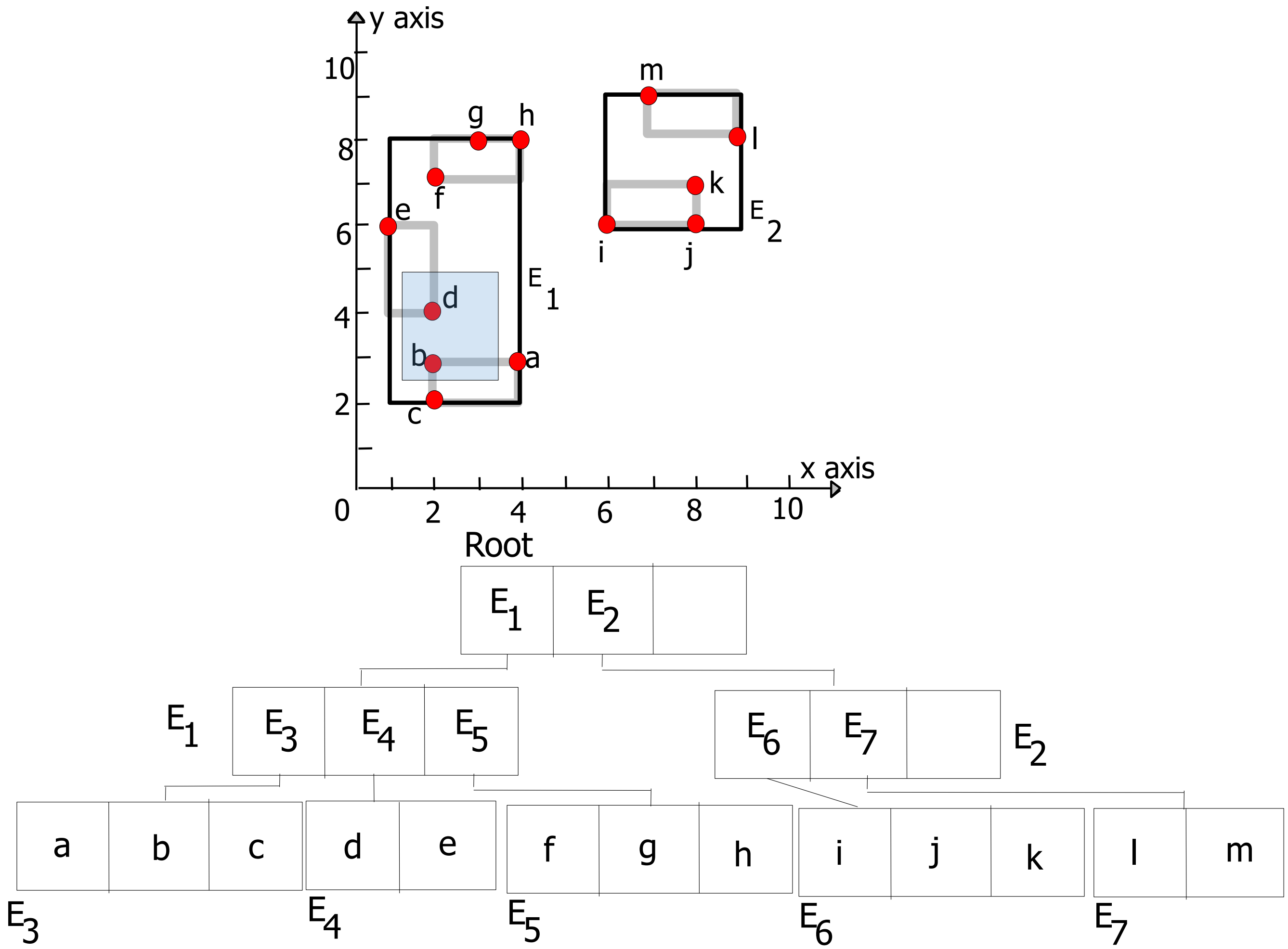
R-Tree, Leaf Nodes



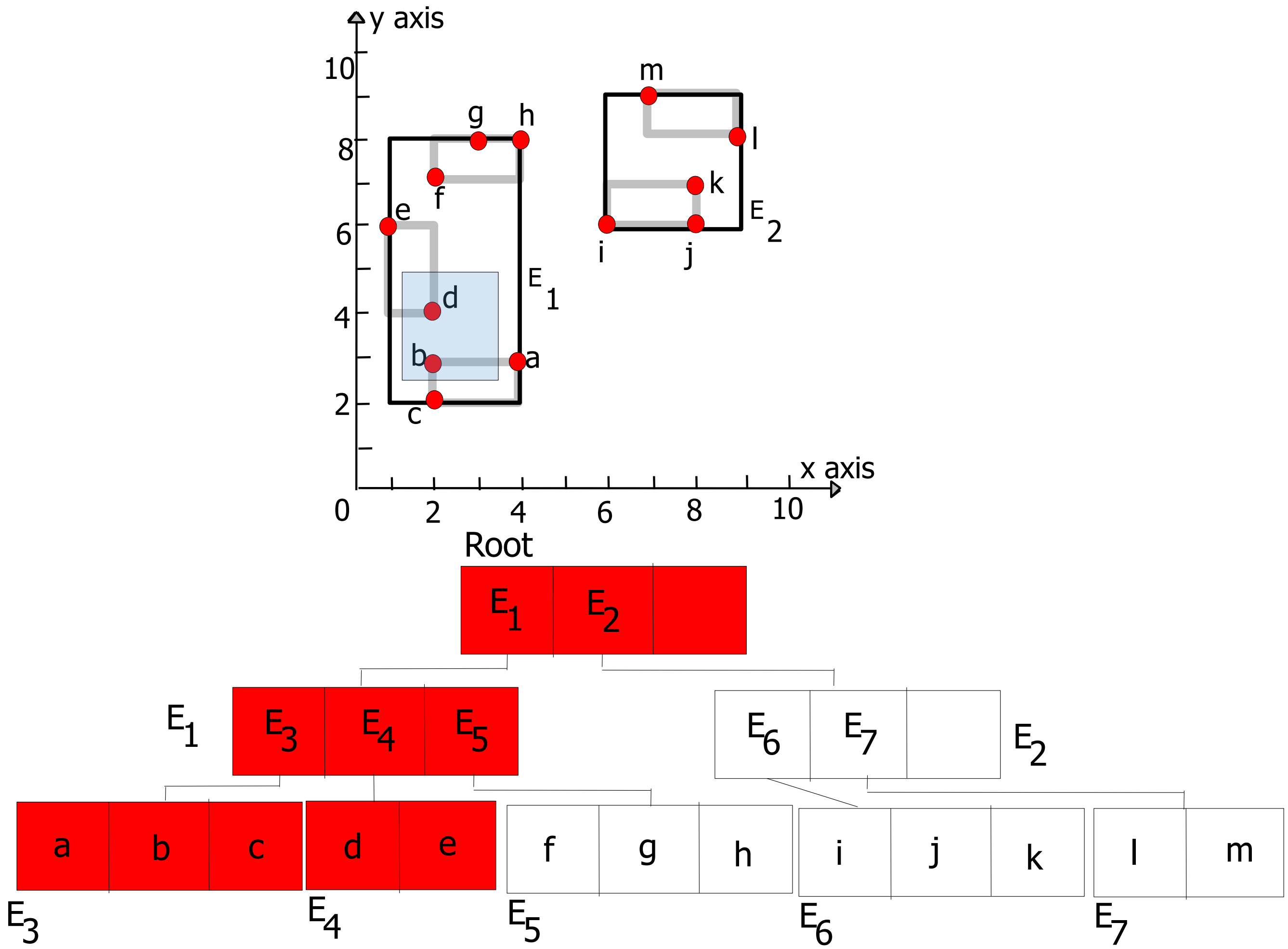
R-Tree – Intermediate Nodes



R-tree, Range Query



Range Query



y axis
10
8
6
4
2
0
2
4
6
8
10
x axis
g
h
f
e
d
b
a
c
E 1
m
l
k
E 2
i
j
Root
E1
E2
E1
E3
E4
E5
E6
E7
E2
a
b
c
d
e
f
g
h
i
j
k
l
m
E3
E4
E5
E6
E7

R-trees: Variations

- R+-tree: DO not allow overlapping, so split the objects (similar to z-values)
- R*-tree: change the insertion, deletion algorithms (minimize not only area but also perimeter, forced re-insertion)

A Theory of Computation

\

- While we have introduced many problems with polynomial-time algorithms…
  - We haven’t formally defined what this means
  - And not all problems enjoy fast computation

- Given a set I of problem instances, and a set S of problem solutions, an abstract problem is a binary relationship in I x S. That is, a set of pairs (i,s)
  - The problem shortest path associates each instance of a graph and an origin-destination with a sequence of vertices which connect the origin and destination

- D**ecision** problems have a yes or no solution. Abstract decision problem is a function which maps problem instances I into {yes, no}.
  - e.g. Is the shortest path in this graph between nodes 0 and nodes 30 have length > 10?

Decision Problem Instance

\

- If a machine is to solve an instance of a problem, the input must be specified in terms of a string of bits
  - We must encode problem instance I into a sequence of binary inputs
  - This helps understand the “size” of the problem instance
  - A concrete problem is an encoding of the set of problem instances to the set of binary strings
  - And not all problems enjoy fast computation

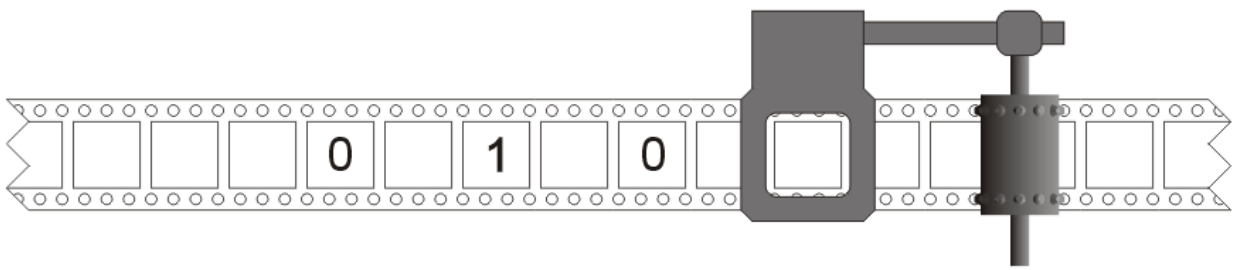
- An algorithm is a procedure that processes an concrete problem instance of size n and generates the correct answer
  - In what computer model?
  - Turing machine (1936) — Alan Turing (developed decoders for German coders (WW II))

Turing machine 1

\

The Turing machine has four components:

- An arbitrary-length tape
- **A head that can**
  - Read a symbol off the tape,
  - Write a symbol to the tape, and/or
  - Move to the next entry to the left or the right

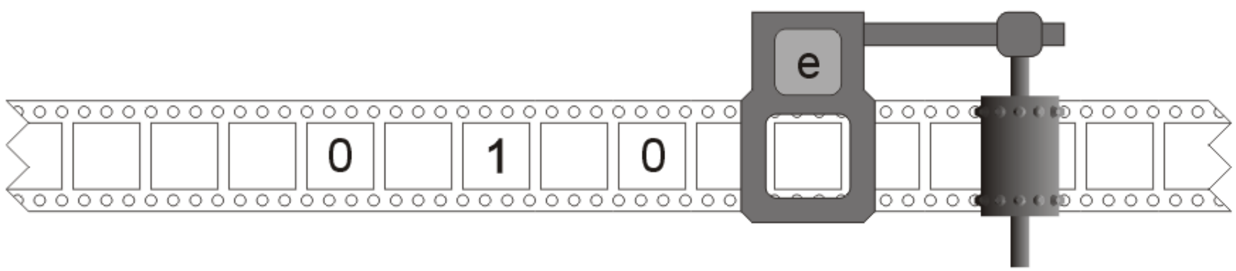


Turing machine 2

\

The Turing machine has four components:

- An arbitrary-length tape
- A head
- **A state**
  - The state is one of a finite set of symbols Q
  - In this example, $\mathrm{Q} = \{b, c, e, f\}$
  - The initial state of the machine is denoted $q_0 \in \mathrm{Q}$
  - Certain states may halt the computation



Turing machine 3

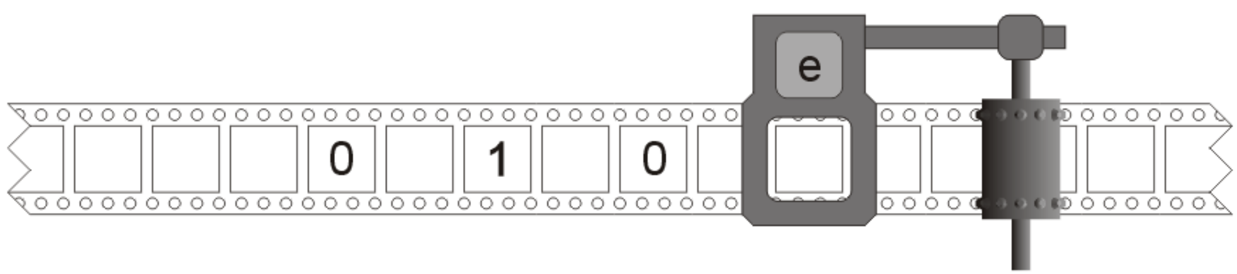
\

The Turing machine has four components:

- An arbitrary-length tape
- A head
- A state
- **A transition table** (this is your program!)
  - $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathbf{L}, \mathbf{R}, \mathbf{N}\}$
  - **L** moves one entry to the left
  - **R** moves one entry to the right
  - **N** indicates no shift

Current		**New**		
State	Symbol read	State	Symbol to write	Direction
b	***B***	*c*	0	**R**
c	***B***	*e*	***B***	**R**
e	***B***	*f*	1	**R**
f	***B***	*b*	***B***	**R**

There is at most one entry in this table for each pair of current settings

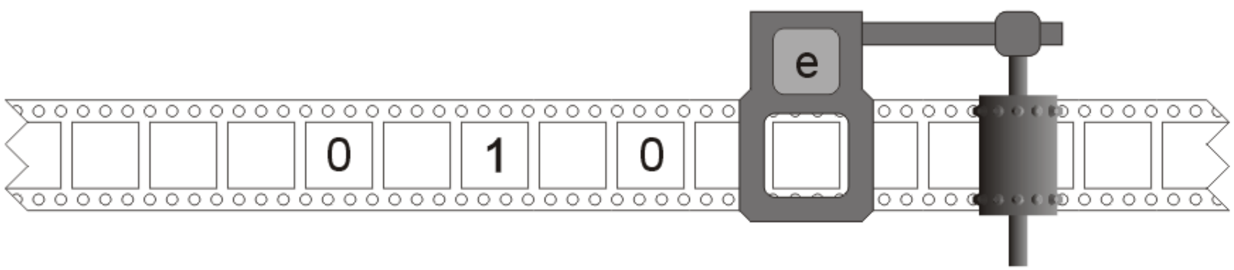


Example (Turing, '36)

A program to write 0 1 0 1 0 1 0 ⋯

Currently, the state is e and the symbol under the head is $\boldsymbol{B}$

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**



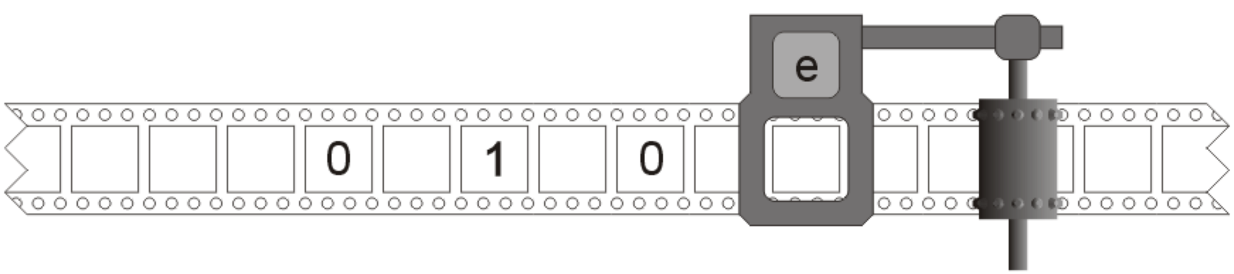
Example - 2

\

The transition table dictates that the machine must:

- The state is set to f
- Print symbol 1 onto the tape
- Move one entry to the right

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	***B***	c	0	**R**
c	***B***	e	***B***	**R**
e	***B***	f	1	**R**
f	***B***	b	***B***	**R**

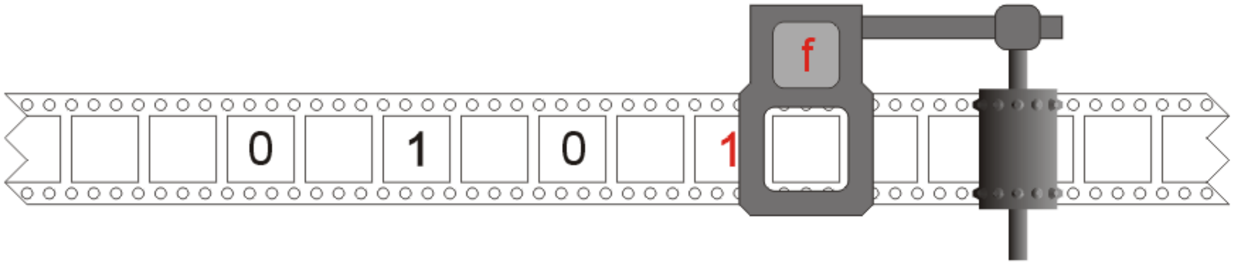


Example - 3

\

The state and symbol under the head have been updated

Current		**Next**		
State	Symbol read	State	Symbol to write	Direction
b	***B***	*c*	0	**R**
c	***B***	*e*	***B***	**R**
e	***B***	*f*	1	**R**
f	***B***	*b*	***B***	**R**



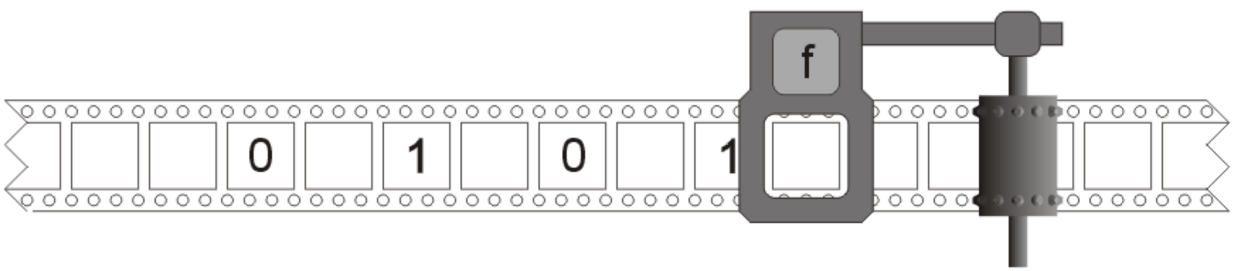
Example - 4

\

The state is f and the symbol under the head is the blank $\boldsymbol{B}$:t

- The state is set to b
- A blank is printed to the tape
- Move one entry to the right

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**



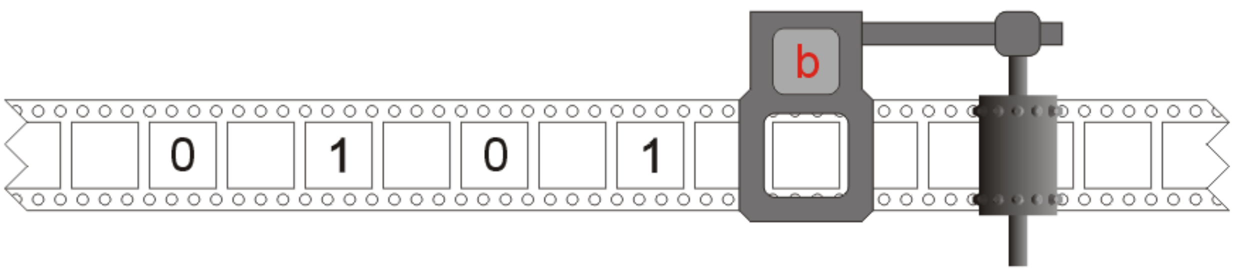
Example - 5

\

Again, the state is b, the symbol a blank, and therefore:

- Set the state to c
- Print the symbol 0 to the tape
- Move one entry to the right

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	***B***	c	0	**R**
c	***B***	e	***B***	**R**
e	***B***	f	1	**R**
f	***B***	b	***B***	**R**



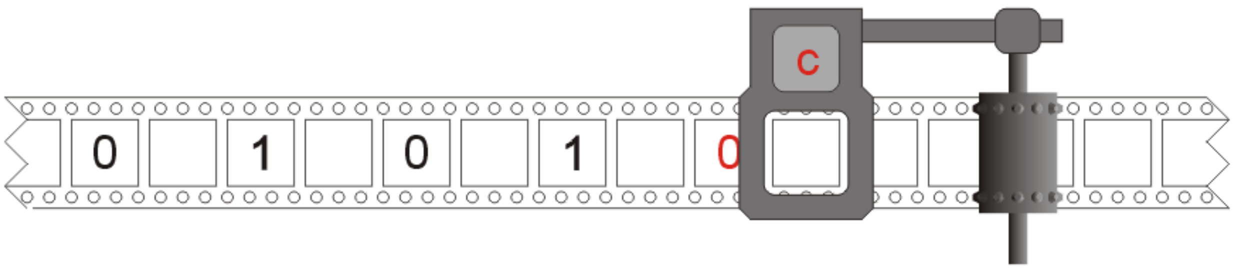
Example - 6

\

The result is the state c and a blank symbol is under the head:

- Set the state to e
- Write a blank to the tape
- Move one entry to the right

Current		**Next**		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**

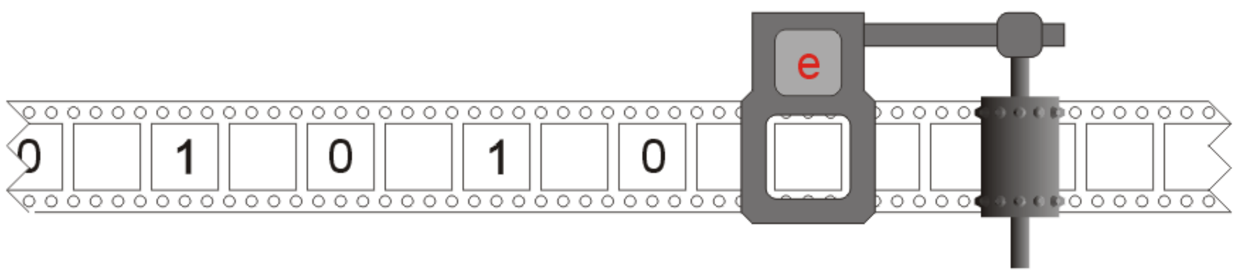


Example - 7

The result is the state e and a blank symbol **B** under the head

- This is the state we were in four steps ago
- This machine never halts...

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**



Another Example

\

This Turing machine does what?

- Tape symbols: $\Gamma = \{\boldsymbol{B}, 1\}$
- States: $Q = \{a, b, c, d, e, \boldsymbol{H}\}$
- Initial state: $q_0 = a$
- Halting state: $\boldsymbol{H}$

Current		Next		
State	Symbol read	State	Symbol to write	Direction
a	$\boldsymbol{B}$	a	$\boldsymbol{B}$	**R**
a	1	b	1	**R**
b	$\boldsymbol{B}$	c	1	**R**
b	1	b	1	**R**
c	$\boldsymbol{B}$	d	$\boldsymbol{B}$	**L**
c	1	b	1	**R**
d	$\boldsymbol{B}$	d	$\boldsymbol{B}$	**L**
d	1	e	$\boldsymbol{B}$	**L**
e	$\boldsymbol{B}$	$\boldsymbol{H}$	$\boldsymbol{B}$	**R**
e	1	e	1	**L**

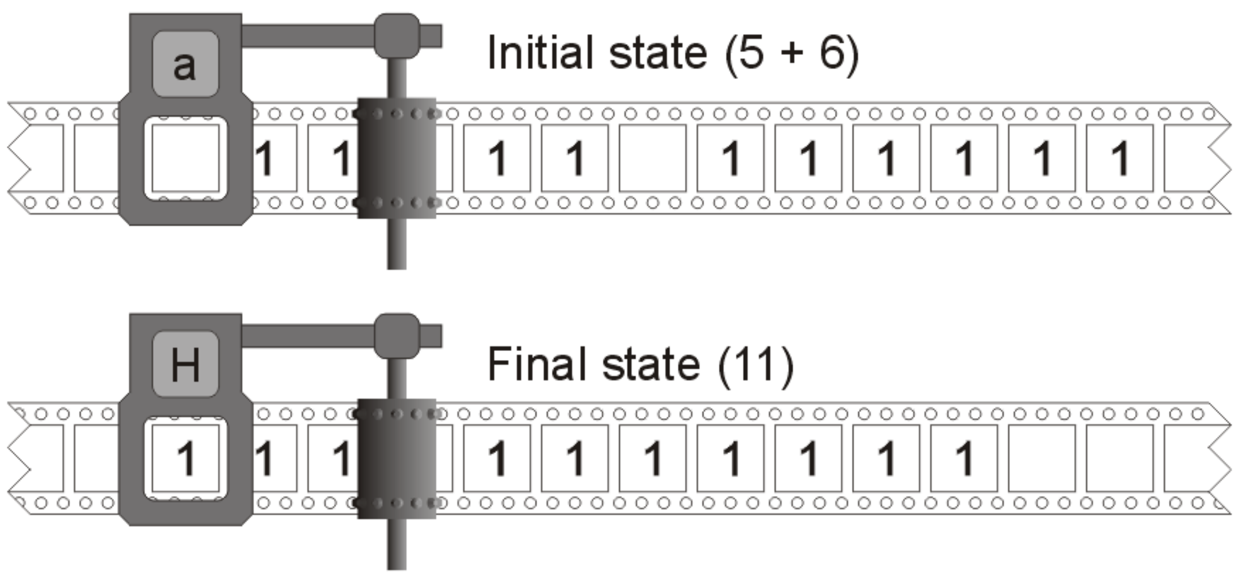
Note there is exactly one entry for each pair in $Q \setminus \{\boldsymbol{H}\} \times \Gamma$

- It may not be necessary to have one for each, but you cannot have more than one transition for a given state and symbol
- **Deterministic** program

Example 2

After 22 steps, a group of five ones and a group of six ones are merged into a single group of eleven ones

- This represents 5 + 6 = 11
- It is the simplest addition machine (no boolean representation of numbers)



Non-deterministic algorithms

A Turing machine is non-deterministic if the transition table can contain more than one entry per state-letter pair

- When more than one transition is possible, a non-deterministic Turing machine branches and creating a new sequence of computation for each possible transition

A non-deterministic algorithm can be implemented on a deterministic machine in one of three manners:

- Assuming execution along any branch ultimately stops, perform a depth-first traversal by choosing one of the two or more options and if one does not find a solution, continue with the next option
- Create a copy of the currently executing process and have each copy work on one of the next possible steps
  - These can be performed either on separate processors or as multiple processes or threads on a single processor
- Randomly choose one of the multiple options

Turing-Church Conjecture

\

Alan Turing and Alonzo Church (Turing’s PhD mentor at Princeton):

- For any algorithm which can be calculated given arbitrary amounts of time and storage, there is an equivalent Turing machine for that algorithm
- Formally: a function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine
  - 'Effective method’: each step of which is precisely predetermined and which is certain to produce the answer in a finite number of steps

A computational system is said to be *Turing complete* if it can compute every function computable on a Turing machine

- e.g., a programming language compiled into machine code and run on a processo

Decision Problem Instance

- An algorithm solves a concrete problem in time O(T(n)) if, when provided with a problem instance of size n bits, it produces the correct answer to the question in O(T(n)) time
- **Polynomially solvable** problems: $T(n) = n^k$ for some k > 0
- Size of the input:
  - Integers represented in binary
  - Sets represented in bits related to the number of elements in the set times the number of bits per element

P and NP

\

- A decision problem belongs to the class **P** if there is a algorithm solving the problem with a running time on a deterministic machine that is polynomial in the input size.

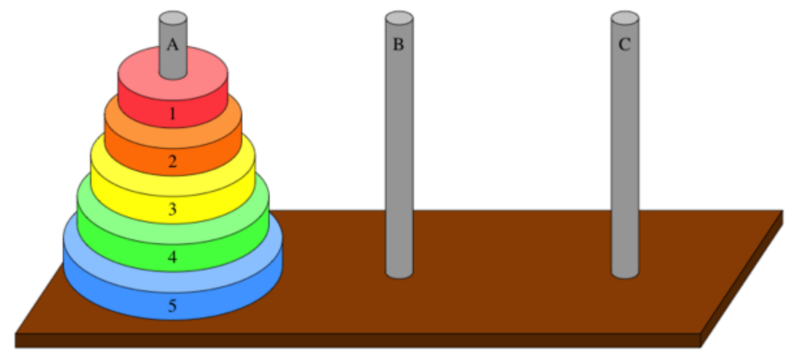
- A decision problem belongs to the class **NP (**non-deterministic polynomial) if:
  - Any solution y leading to ‘yes' can be encoded in polynomial space with respect to the size of the input x.
  - Checking whether a given solution leads to `yes' can be done in polynomial time with respect to the size of x and y
  - problem is solvable in polynomial time in a non-deterministic machine
    - Can explore all possible solutions in parallel
    - Oracle selects best possible solution to check

<u>Slightly more formal:</u>

- Problem Q belongs to the class **NP**, if there exists a polynomial-time 2-argument algorithm A, such that:
  - For each instance i, i is a yes-instance to Q, if and only if, there is a polynomial-size certificate c for which A(i,c) = true.

Examples

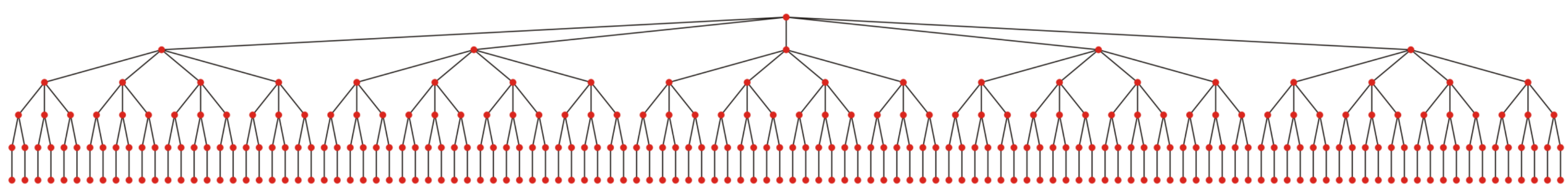
\



- Tower of Hanoi
  - Number of moves to solve is exponential in number of disks
  - Can’t describe solution in polynomial number of moves
  - Not in NP
- Shortest path problem
  - Is the length of a shortest path from a to b less than a threshold?
  - Can answer in polynomial time in the size of the description of the network —> It is in **P**
- Traveling salesperson problem
  - Is the length of a complete traveling salesperson tour less than a threshold?
  - A tour can be described in polynomial space, and we can verify the length of the tour in polynomial time —> it is in **NP**

Non-deterministic polynomial-time algorithms

- The traveling salesman problem can solved non-deterministically:
  - At each step, spawn a thread for each possible path
  - As you finish, compare them and determine if any of them have length less than k
  - The run time is now $\Theta(|V|)$
  - This is a brute-force search



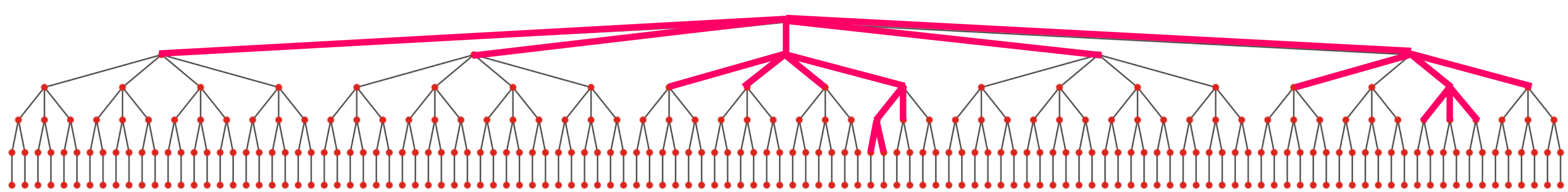
Non-deterministic polynomial-time algorithms

Consider the following decision problem:

“Is there a path between vertices a and b with weight no greater than K?”

Dijkstra’s algorithm can answer this in polynomial time

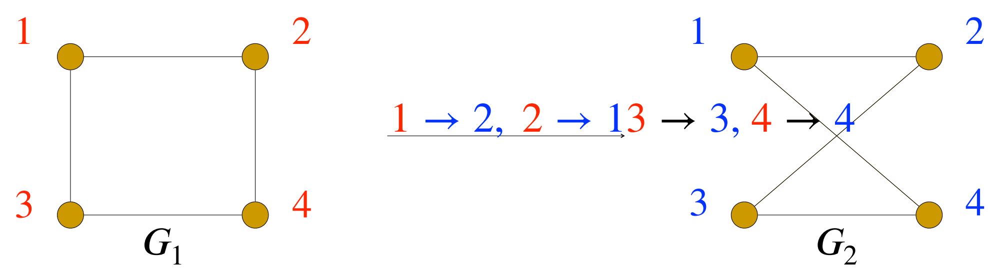
- Dijkstra’s algorithm also solves the optimization problem



Examples of $\mathbf{NP}$ problems

- Factoring: factor a given number n.
- Decision version: Given (n, k), decide whether n has a factor less than k
- Factoring is in **NP**: For any candidate factor $m \le k$, it's easy to check whether $m \mid n$.

- Graph Isomorphism: Given two graphs G_1 and G_2, decide whether we can permute vertices of G_1 to get G_2.



- Easy to check: For any given permutation, easy to permute G_1 according to it and then compare to G_2.

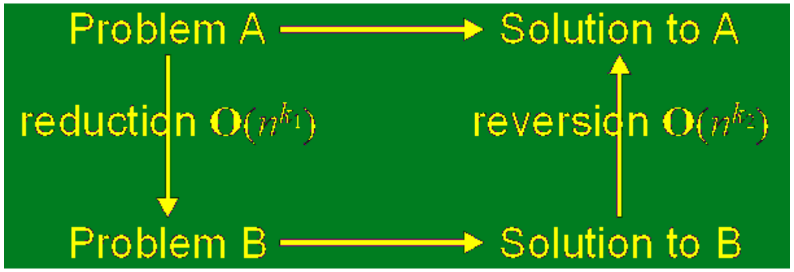
Reduction and completeness

- Decision problem for language A is reducible to that for language B in time t if $\exists f: Domain(A) \to Domain(B)$ s.t. $\forall$ input instance x for A,
  1. $x \in A \Leftrightarrow f(x) \in B$, and
  2. one can compute $f(x)$ in time $t(|x|)$
- Thus to solve A, it is enough to solve B.
  - First compute $f(x)$
  - Run algorithm for B on $f(x)$.
  - If the algorithm outputs $f(x) \in B$, then output $x \in A$.

Reduction

- Reduction converts the solution of a problem to the solution of another problem

- Graphically, we may think of the following image:



- To solve Problem A, we:
  - Reduce the problem to Problem B in polynomial time
  - Solve Problem B
  - Revert the solution back into a solution for Problem A

- We want the reduction and reversion algorithms to be of polynomial complexity: **polynomial reduction**

Example: Polynomial reduction

\

- Multiply two n digit decimal numbers:
  - Reduction: convert the two numbers into binary numbers
  - Multiply the two binary numbers
  - Reversion: convert the solution back into a decimal number

- Both the reduction and the reversion run in $\Theta(n)$ time

- Observe: if a decision problem is reduced to a decision problem, the corresponding reversion algorithm is trivial and is in $\Theta(1)$ time

Polynomial reduction

\

- Another example: Does a list have a duplicate element?
  - Reduction: Sort the list
  - Simpler problem: Does a sorted list have a duplicate element?
  - Reversion: Return `true` or `false`, as is

- Both the reduction and the reversion run in $\Theta(n)$ time
  - If a decision problem is reduced to a decision problem, the reversion is therefore $\Theta(1)$
    - Either the solution or its negation

Examples: Polynomial reduction

\

- Example: Does an n by n assignment problem have minimum cost less than K?
  - Polynomial Reduction: Reduce to the solution of n sequential shortest path problems in non-negative weight graphs with O(2n) vertices
  - Reversion: Convert the decision of the successive shortest path algorithm

- Example: Given two sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, is there a permutation j(i) such that $\sum_{i=1}^{n} a_i b_{j(i)} \geq K$?
  - Polynomial Reduction: sort both sequences increasing order, multiply the sorted sequences and verify product is greater than or equal to K
  - Another polynomial reduction: convert to assignment problem!

Polynomially Reducible

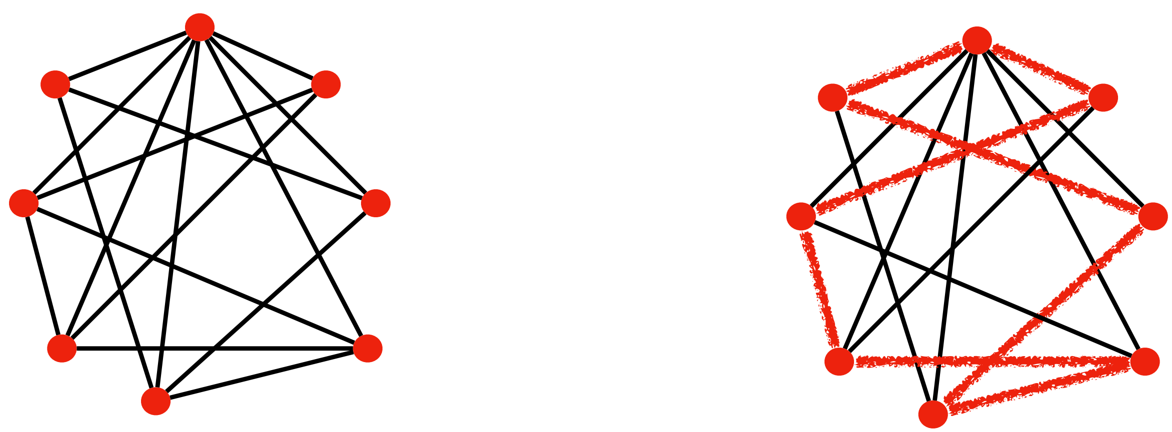
\

- Definition: Problem A is polynomially reducible to problem B if there exists an algorithm for solving problem A in polynomial time if we could solve arbitrary instances of problem B at unit cost
  - Written as $A \leq_P B$
  - If $A \leq_P B$, and $B \leq_P A$, then we write $A =_P B$

- If $A \in \mathbf{P}$ and $B \leq_P A$, then $B \in \mathbf{P}$
  - e.g. Shortest path problem is in **P**. Assignment problem is polynomially reducible to shortest path problem. Then Assignment problem is in **P**.

Polynomial Reduction

- Problem A: Traveling salesperson problem
  - Given a weighted directed graph, find a simple cycle that visits each vertex once and has total cost less than or equal to K
- Problem B: Hamiltonian cycle problem
  - Given a directed graph, does there exist a simple cycle that includes every vertex once?



Polynomial Reduction

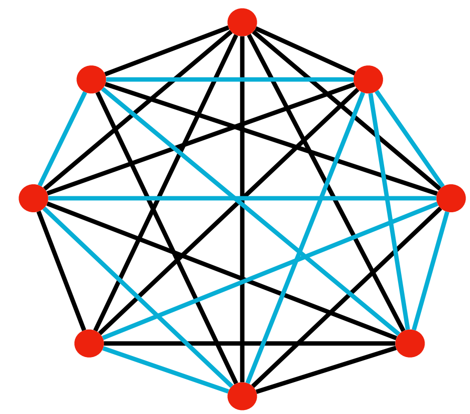
\

- Claim: $B \leq_P A$

Proof: Let graph for B be G(V,E). We are going to construct a weighted graph G(V,E') with a weight function for Problem A, in polynomial time

Let E' be the dense set of edge in $V \times V$. Assign weights as follows: w(e') = 0 if $e' \in E$, w(e') = 1 if $e' \notin E$

Reduction: $O(|V|^2)$ is polynomial



TSP problem: Does there exist a simple cycle that visits every vertex in V once, and has total cost less than or equal to 0

If yes, it is a cycle with all edges in E, and so it is a Hamiltonian cycle

If no, no cycle exists with all edges in E, so no Hamiltonian cycle can be found

This shows Hamiltonian cycle $\leq_P$ Traveling Salesperson Problem