University of BRISTOL | Department of Computer Science

PROGRAMMING and ALGORITHMS II

Algorithms Lecture 1:

DIVIDE & CONQUER

Geometry-centric Intro to a Key CS Paradigm

Dr Tilo Burghardt

Unit Code COMS10001

Babylonian Library Collections

University of BRISTOL | Department of Computer Science

- Finding information fast is an essential task, now as it has been 2000 years ago…
- libraries order their books
- How can one find a book in an ordered array fast?

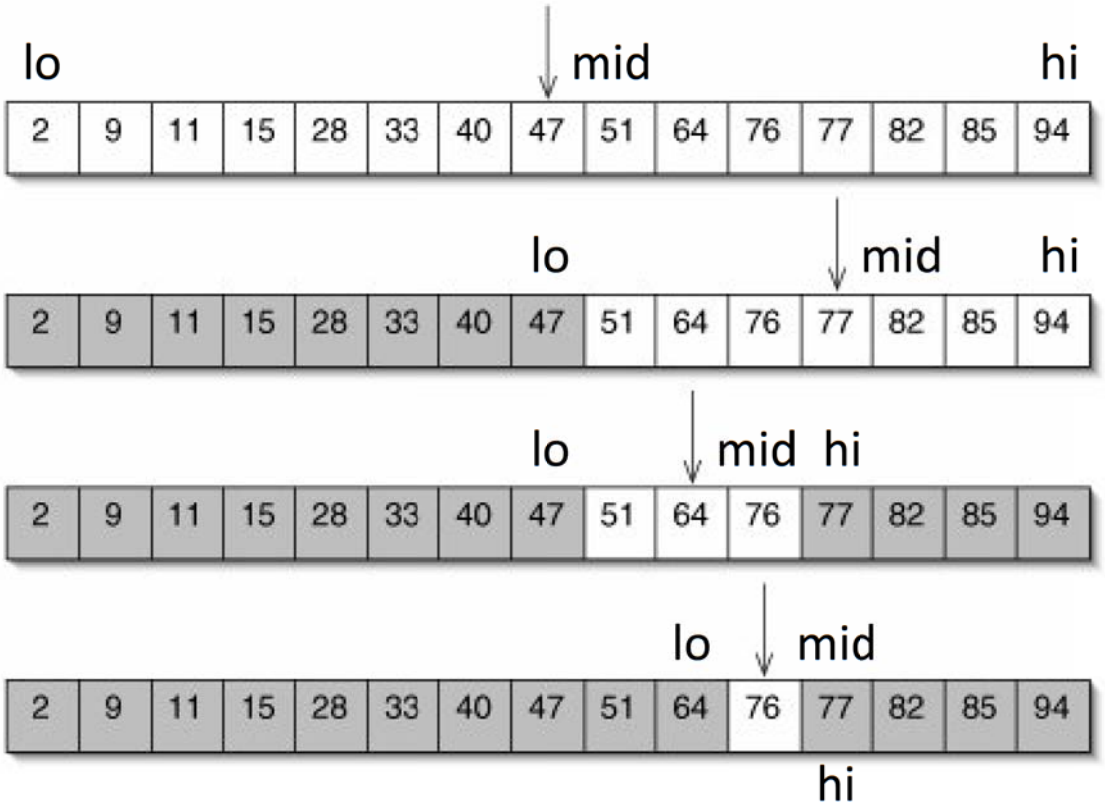
More Formalised Search Problem

Input: query x and sorted array $a[0..n-1]$ where $a[i] \leq a[j]$ and $0 \leq i \leq j < n$

Problem: for the query x find an index i such that $0 \leq i < n$ and $a[i] = x$ or, if such an i is not present, report failure

Finding `tablet' in a sorted array a in *O(log n)* (known since at least 200 BC)

Example: finding table 76



Simple Binary Search Algorithm

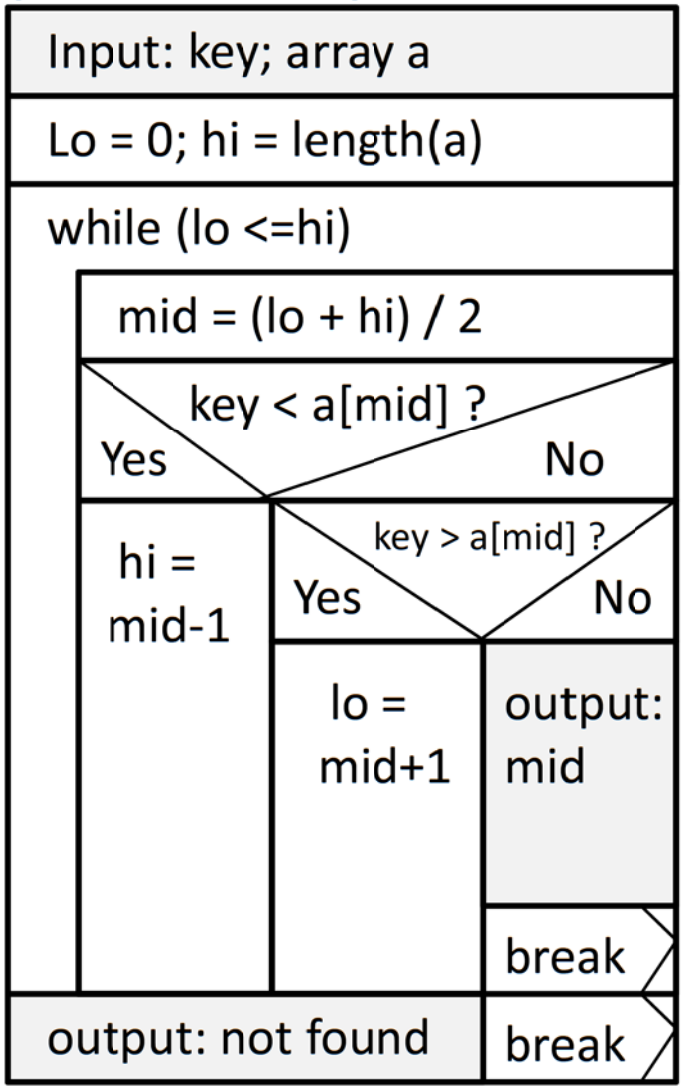
Code Fragment (Java)

```
class Library {
  int[] a; //book keys
  ...
  int find(int key) {
    int lo = 0;
    int hi = a.length - 1;
    while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      //discard upper array part
      if (key < a[mid]) hi = mid - 1;
      //discard lower array part
      else if (key > a[mid]) lo = mid + 1;
      //key found
      else return mid;
    }
    //key not in array
    return -1;
  }...}
```

Shneiderman

Nassi

Nassi-Shneiderman-Diagram (structure chart)



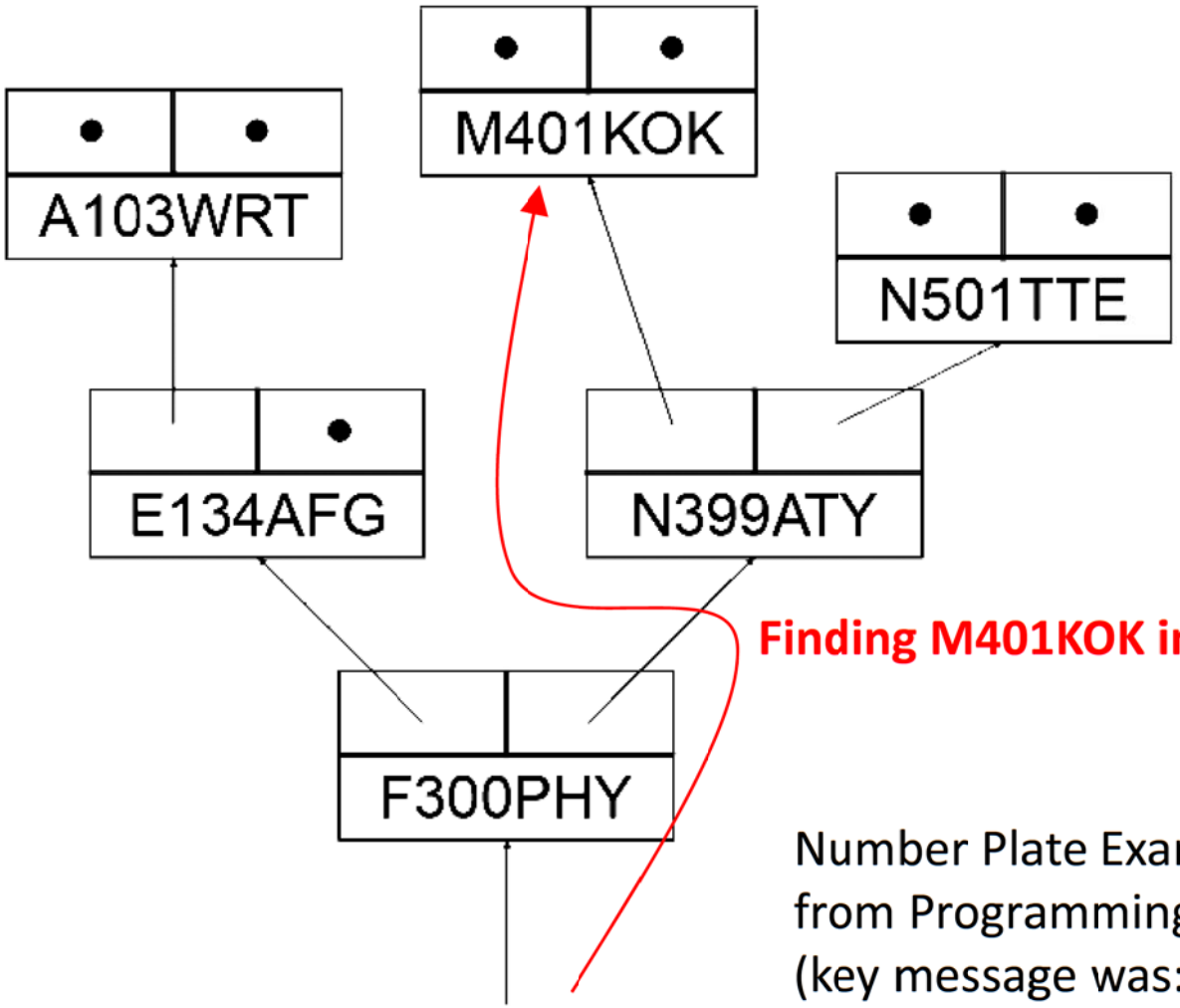
Utilisation and Amortisation

The **actual algorithm utilisation** can heavily impact on the runtime performance of programs.

- Linear Search in unsorted list: $O(n)$
- Binary Search in sorted list: $O(\log n)$
- Sorting the list via Quicksort: $O(n \log n)$

→ amortisation of sorting a list (before searching) from k searches onwards once $kn > n \lg n + k \lg n$

Recap: Binary Search Trees (BSTs)



Finding M401KOK in 3 Steps

Number Plate Example
from Programming and Algorithms I
(key message was: tree search in ***O(lg n)***)

Algorithm Design Paradigm 1

DIVIDE & CONQUER (D&C)

Concept: Recursively *break down* a problem *into independent sub-problems* of related type and smaller size until these become simple enough to be solved directly.

Special Case: If only a single, simpler sub-problem is generated per step (e.g. binary search) the paradigm is also known as *`decrease & conquer'*...

DIVIDE and CONQUER in four phases

1) **PREPARE**
   Transforming the problem into a form suitable for structured subdivision
2) **DIVIDE**
   Recursively breaking the problem into sub-problems that are similar to the original problem but smaller in size,
3) **SOLVE**
   Compute solutions for the sub-problems successively and independently,
4) **CONQUER**
   Combine these solutions to create one solution to the original, larger problem.

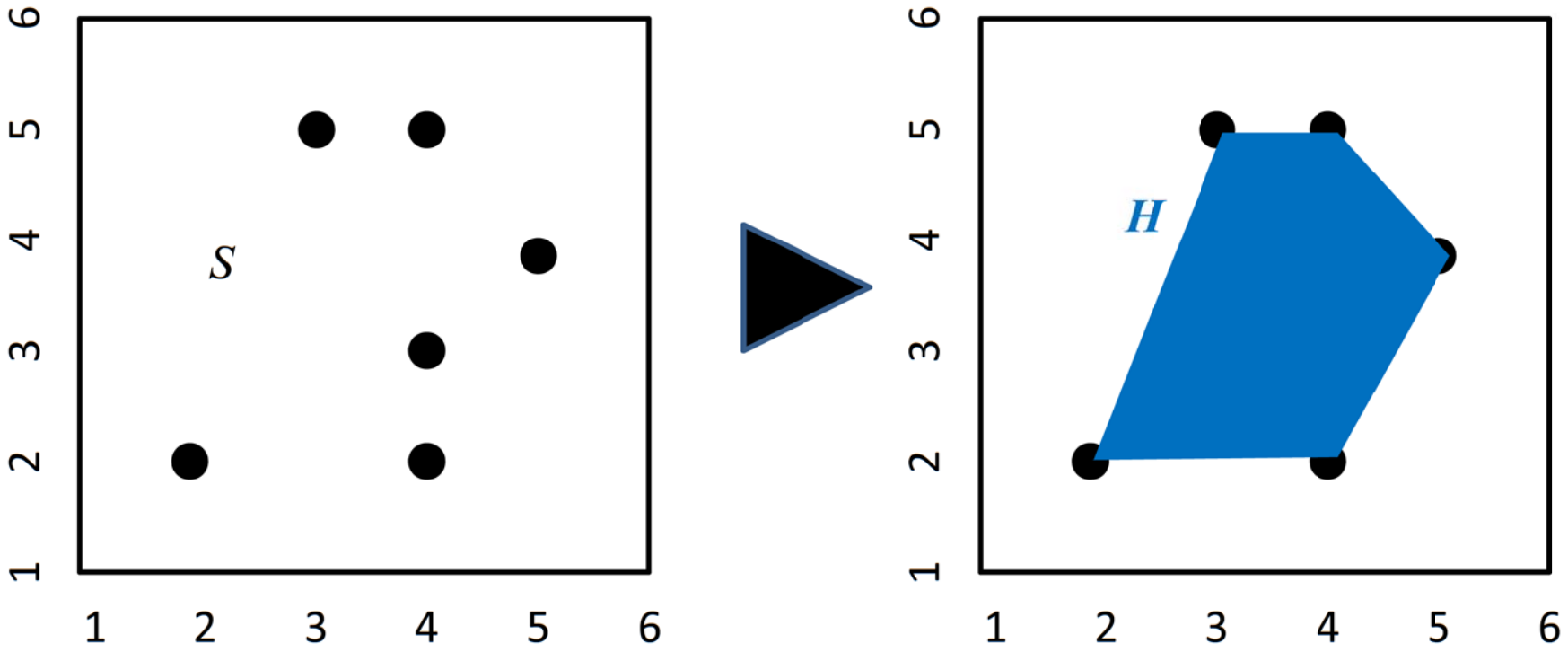
The Convex Hull Problem

?

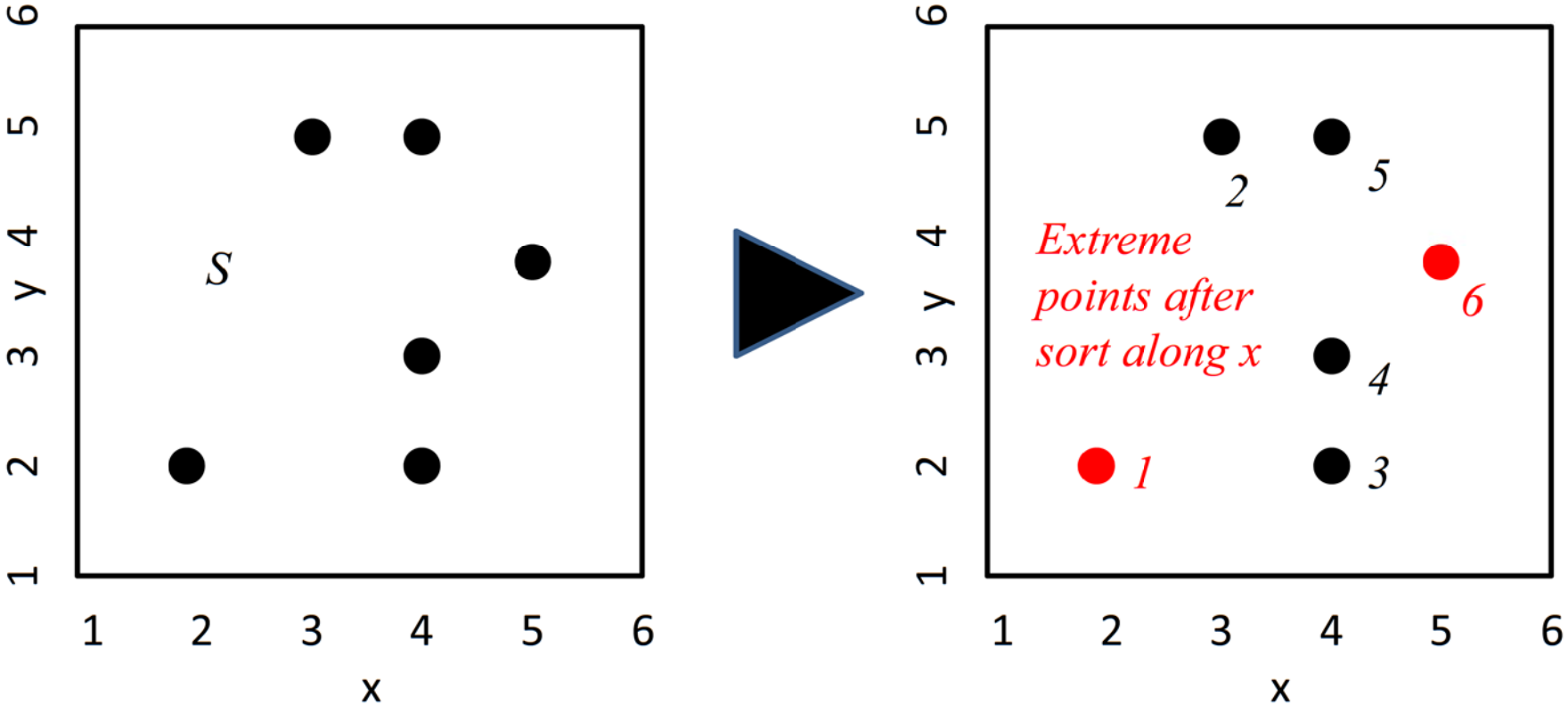
The Convex Hull Problem

Input: Finite 2D Point Cloud, e.g. $S = \{ (3,5) , (4,3) , (5,4) , (4,5) , (2,2) , (4,2) \}$

Problem: Find convex hull $H(S)$, characterised by the circumscribing polygon.

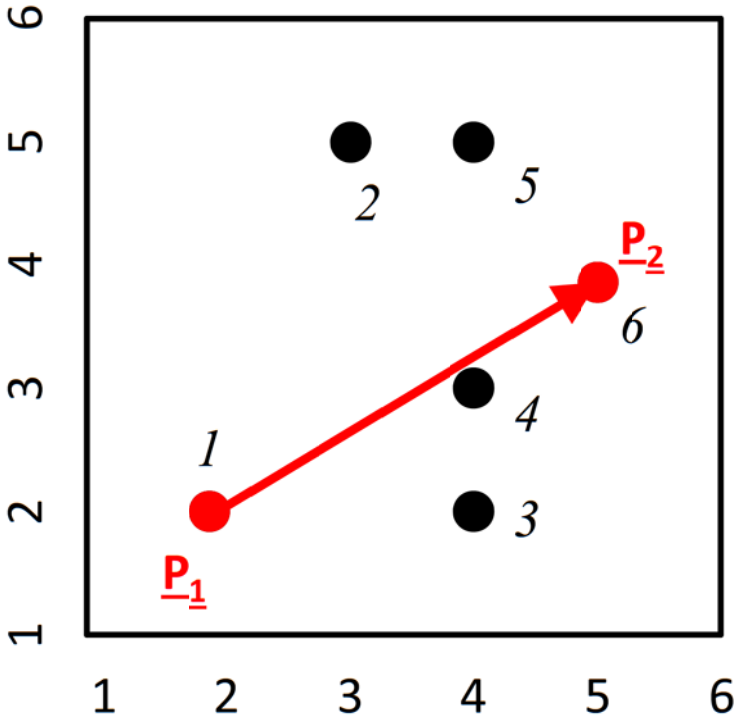


First **sort** 2D Point Cloud S by one dimension, say x, using Quicksort (see P&A1) in $O(n \lg n)$ into a sorted list $((2,2) , (3,5) , (4,2) , (4,3) , (4,5) , (5,4))$.



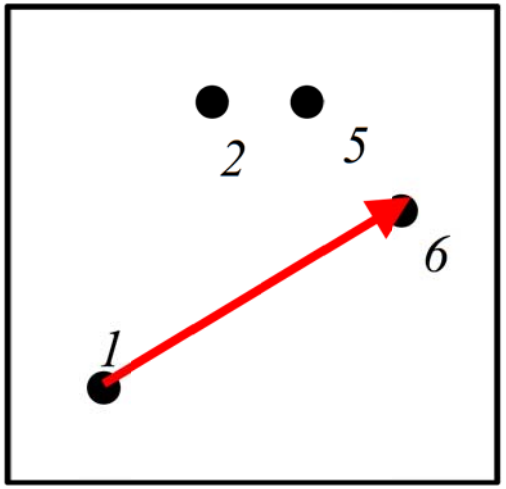
Divide the Problem

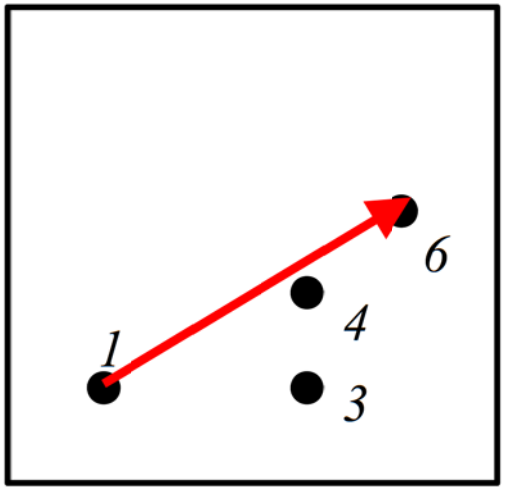
Use vector P_1P_2 defined by extreme points w.r.t. x to split the point cloud (and thereby the problem space) into two smaller problems.



DIVIDE

Calculate Left Hull for points left of P_1P_2

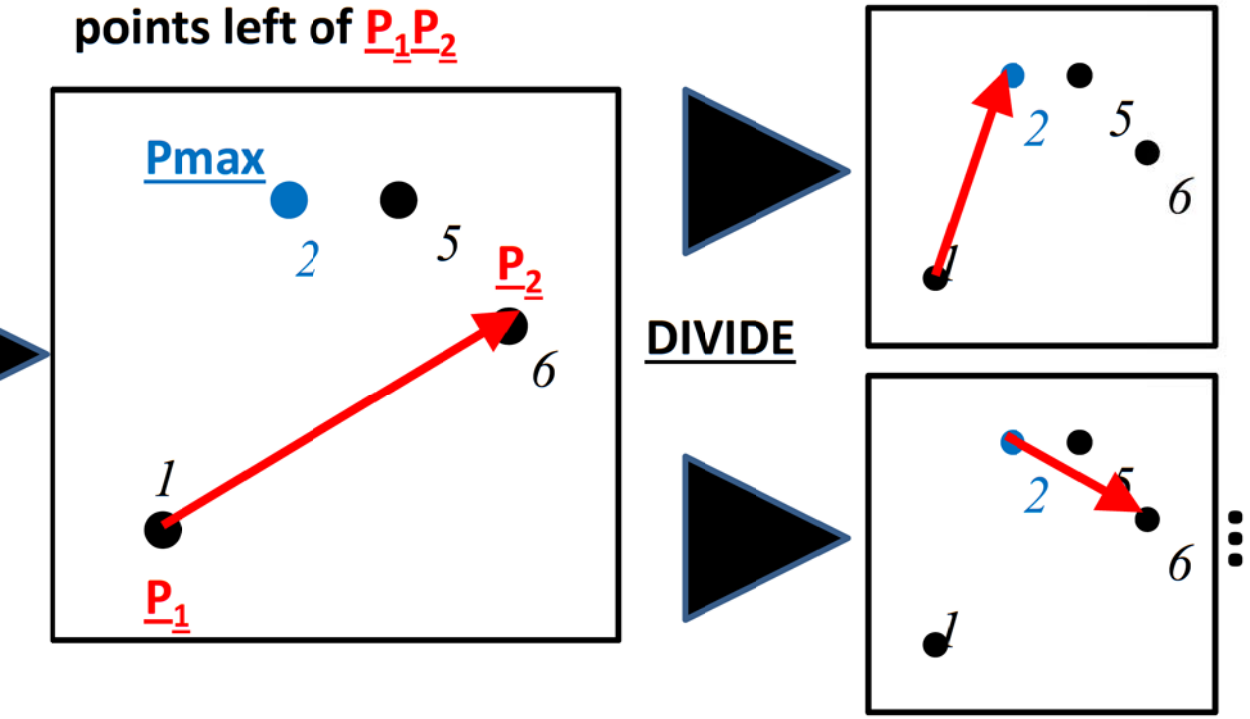




Calculate Right Hull for points right of P_1P_2

Computing Step for the Left Hull

Calculate hull for remaining points left of P_1P_2



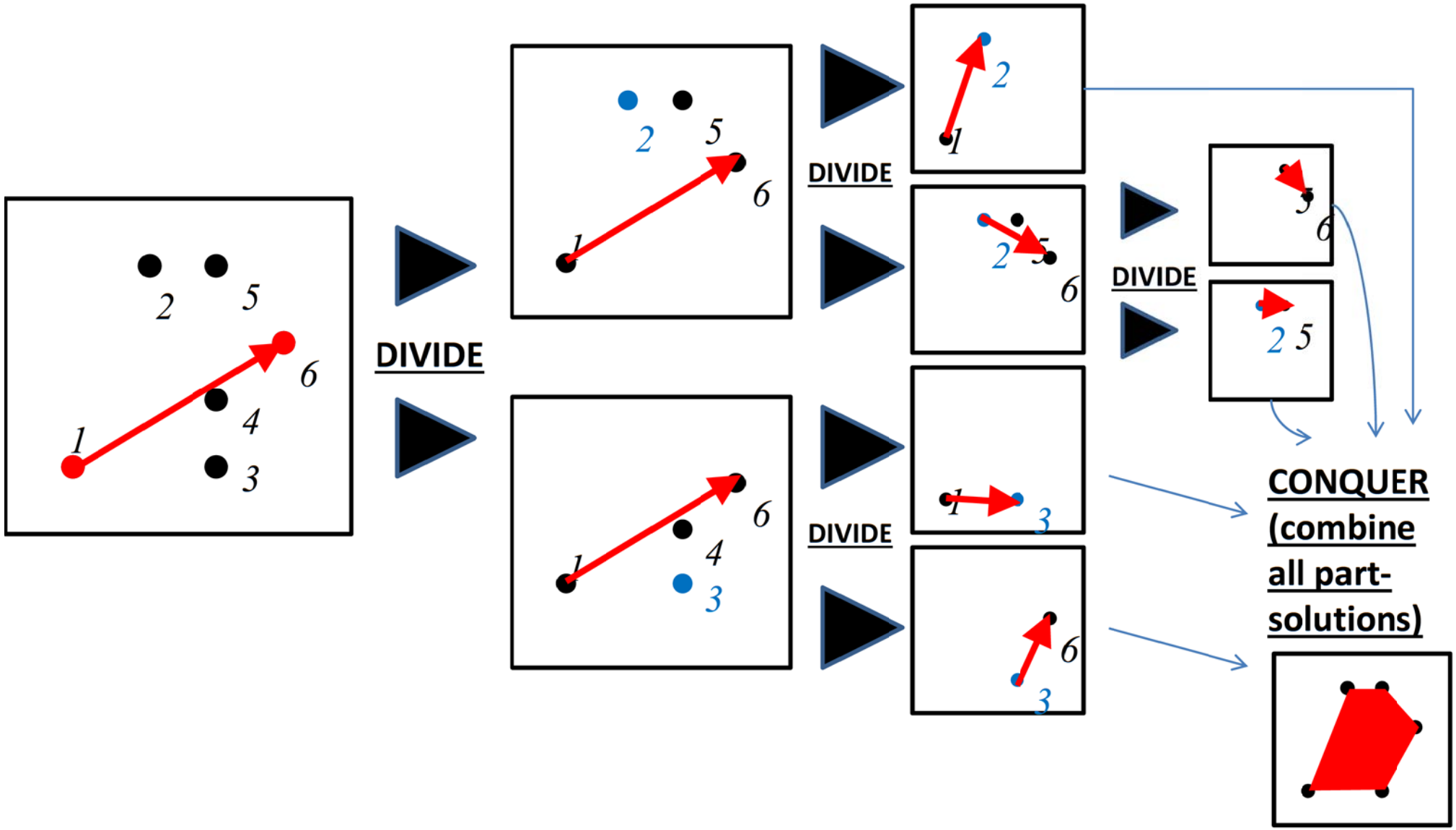
...

For each subproblem determine (if existing) a remaining point Pmax furthest away from the current split vector P_1P_2.

Now divide the problem again by computing the hull for two new split vectors, that is cosider:
A) points left of P_1P_{max} and
B) points left of $P_{max}P_2$

(Compute Right Hull in respective manner considering points right of split vectors.)

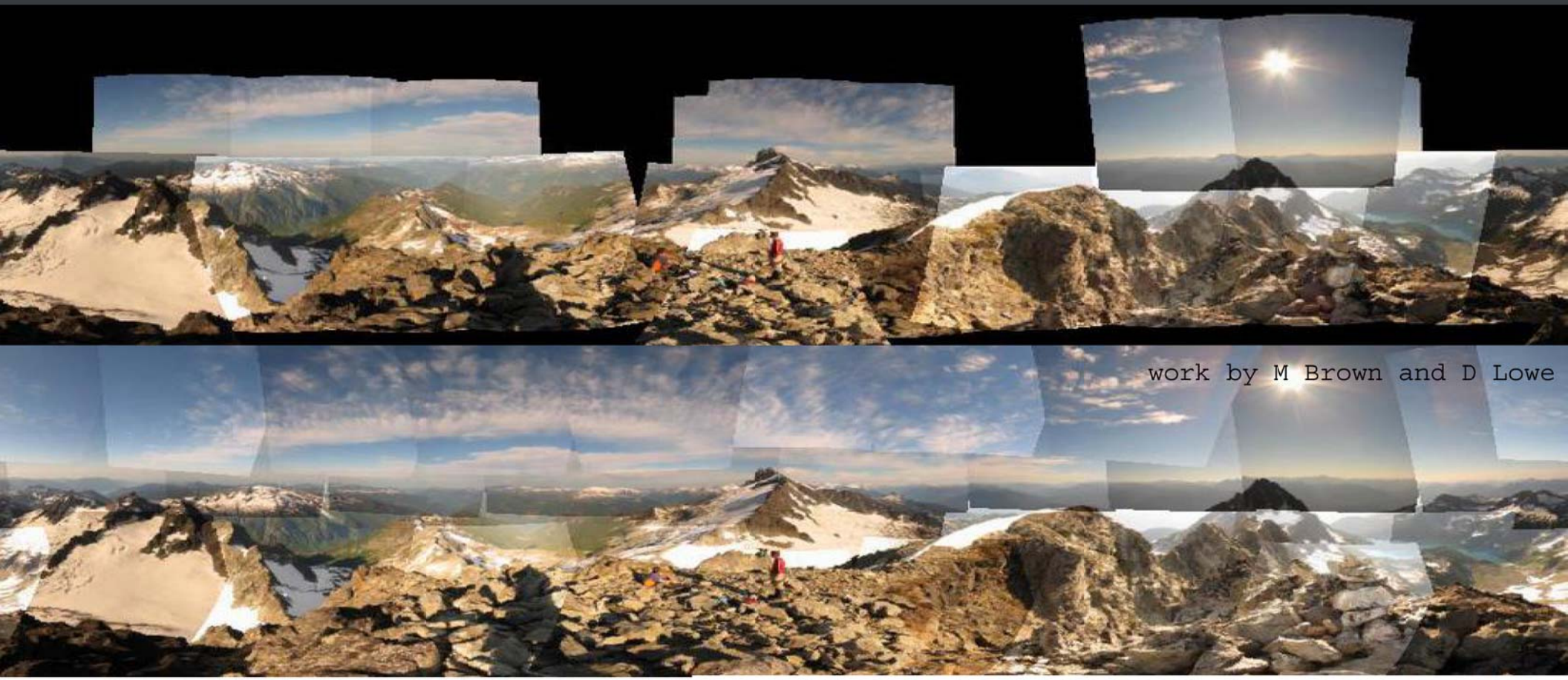
Divide, Solve and finally Conquer!



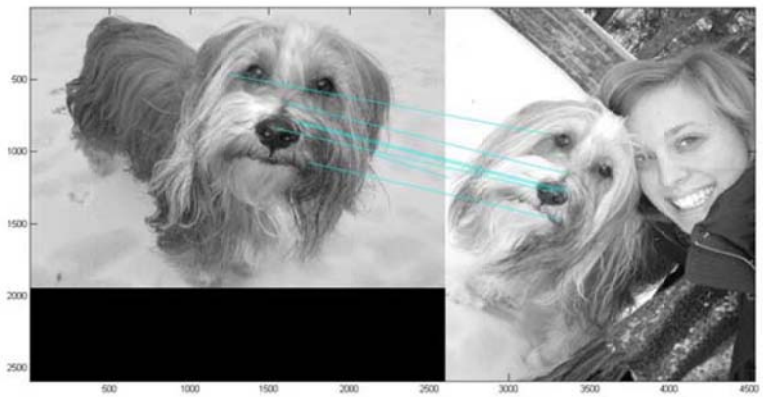
A Real World Application Problem:

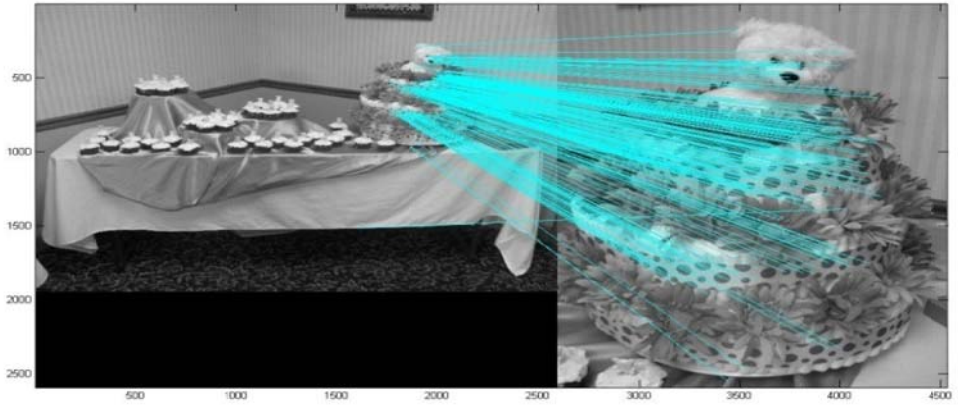
Image Region Correspondence

University of BRISTOL | Department of Computer Science



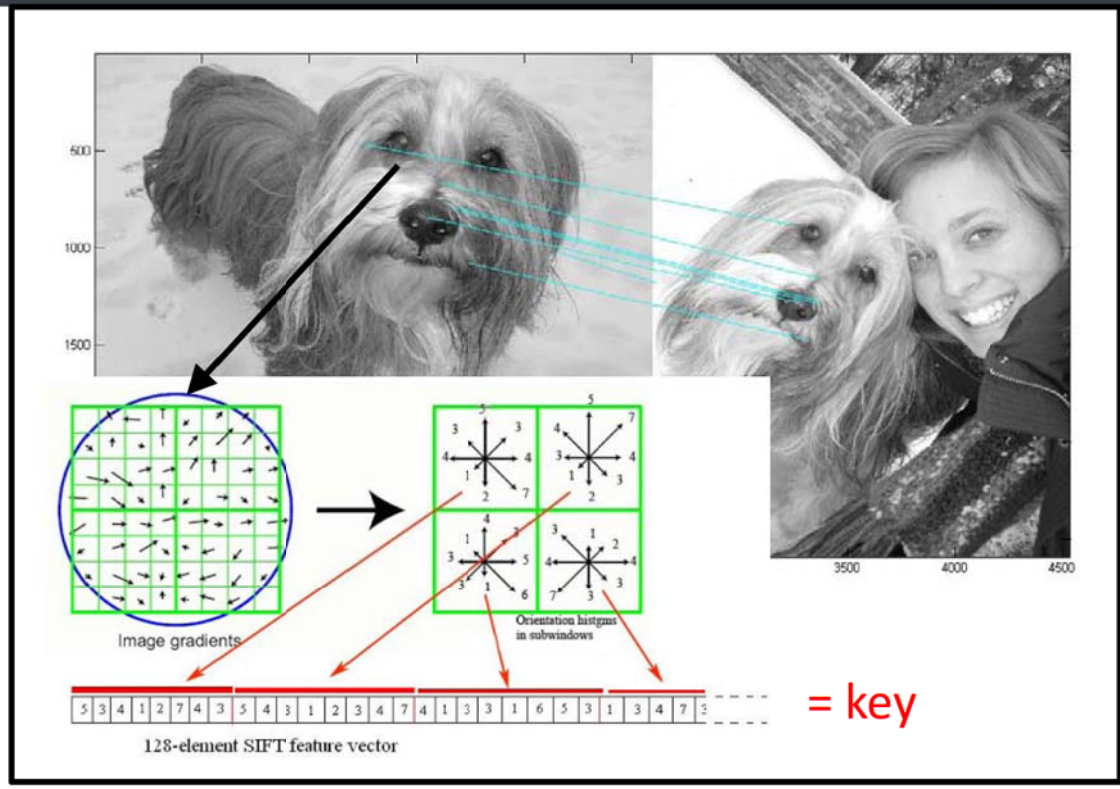
work by M Brown and D Lowe





BSTs for *k* Dimensions

- per location a key vector is formed that characterises *k* different properties of a small region;
- Large numbers of keys need to be stored in an efficient manner; usually via a k-dimensional Binary Search Tree or **kD-Tree**

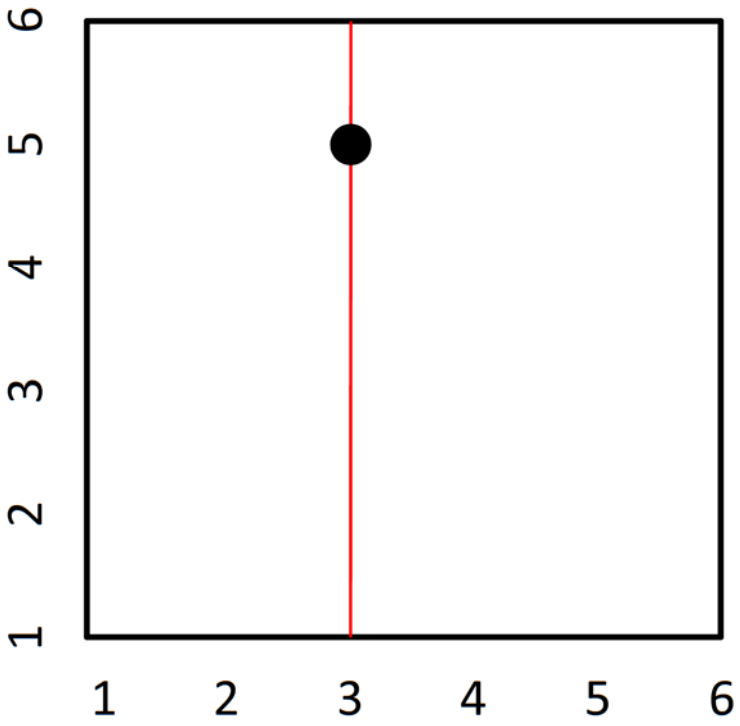


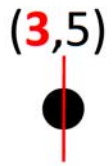
- corresponding keys from a new image can be determined by searching for them or their nearest neighbours (i.e. for similar properties) via the kD-Tree

How can we built a kD-Tree? How can we find keys in it?

Example 2D-Tree Construction

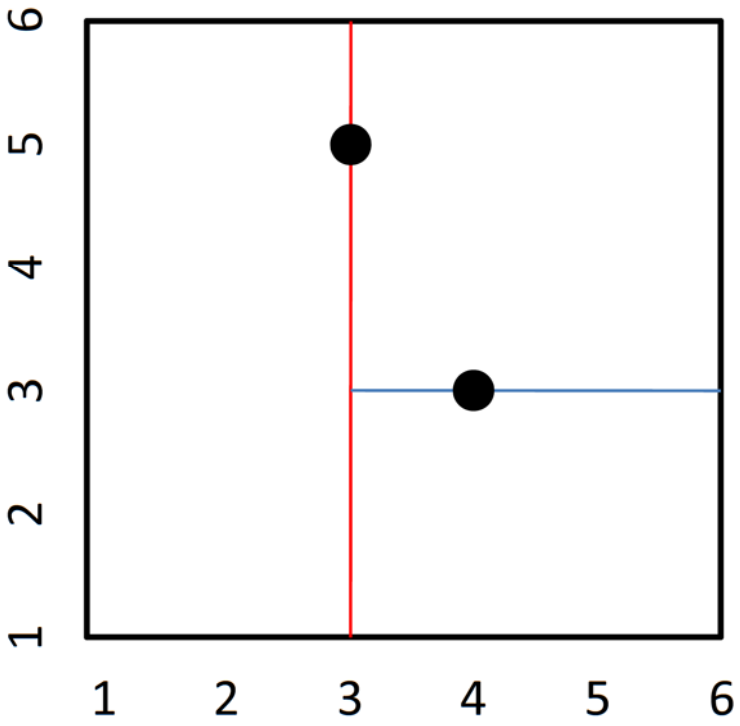
Input Sequence: (3,5) (4,3) (5,4) (4,5) (2,2) (4,2)

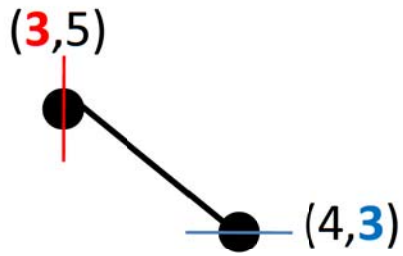




Example 2D-Tree Construction

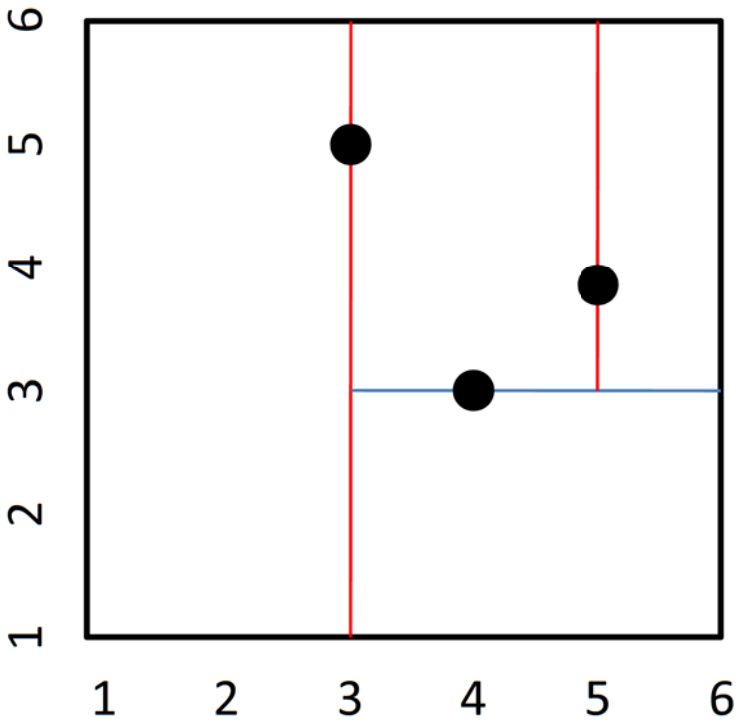
Input Sequence: (3,5) (4,3) (5,4) (4,5) (2,2) (4,2)

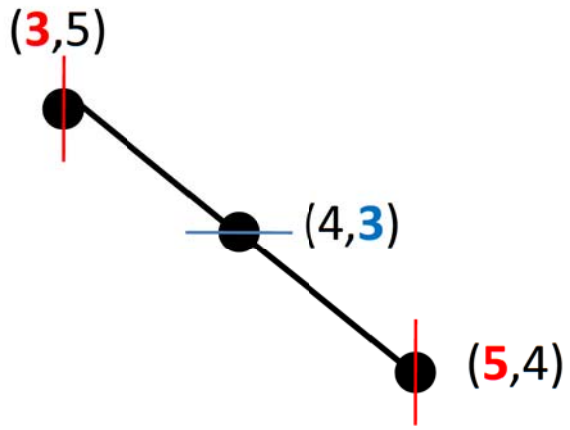




Example 2D-Tree Construction

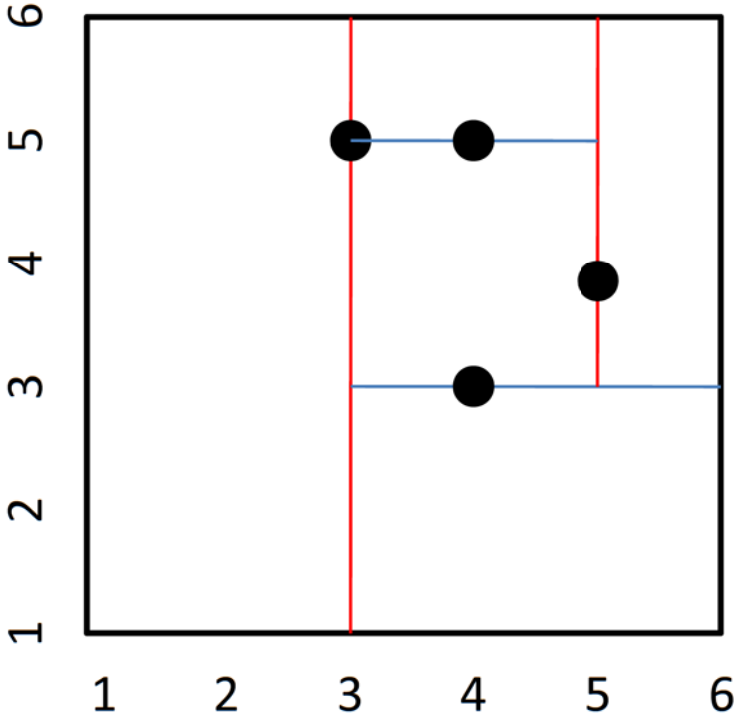
Input Sequence: (3,5) (4,3) (5,4) (4,5) (2,2) (4,2)

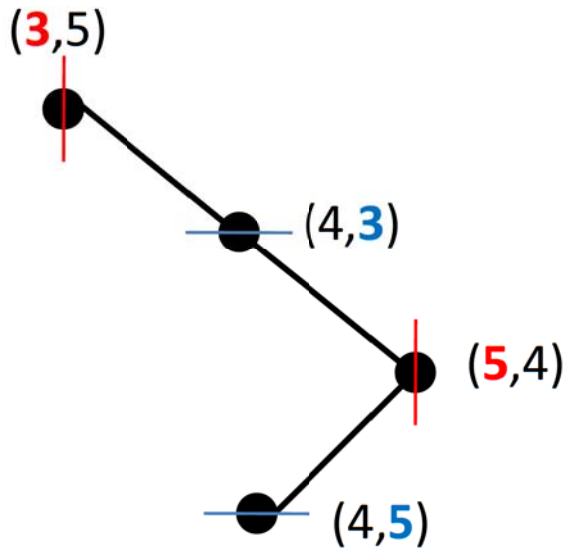




Example 2D-Tree Construction

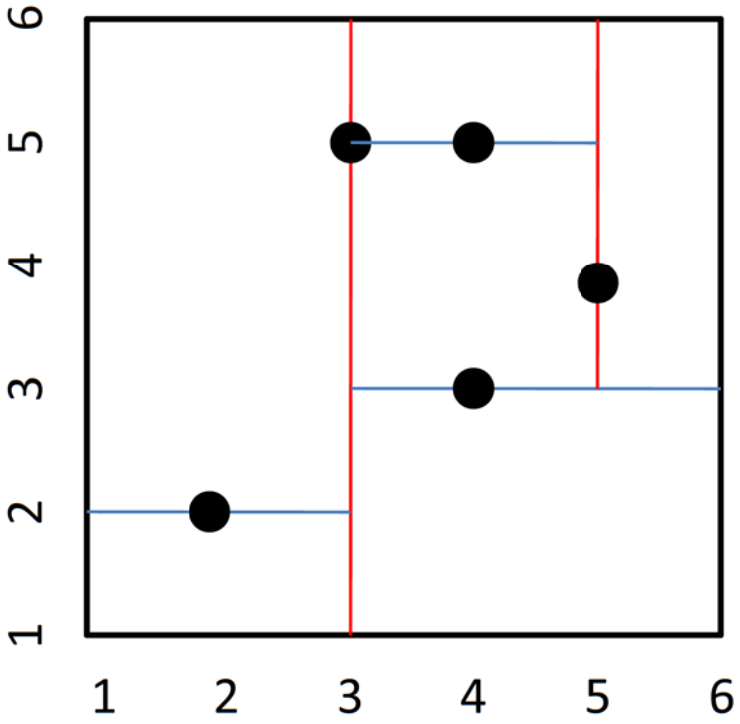
Input Sequence: (3,5) (4,3) (5,4) (4,5) (2,2) (4,2)

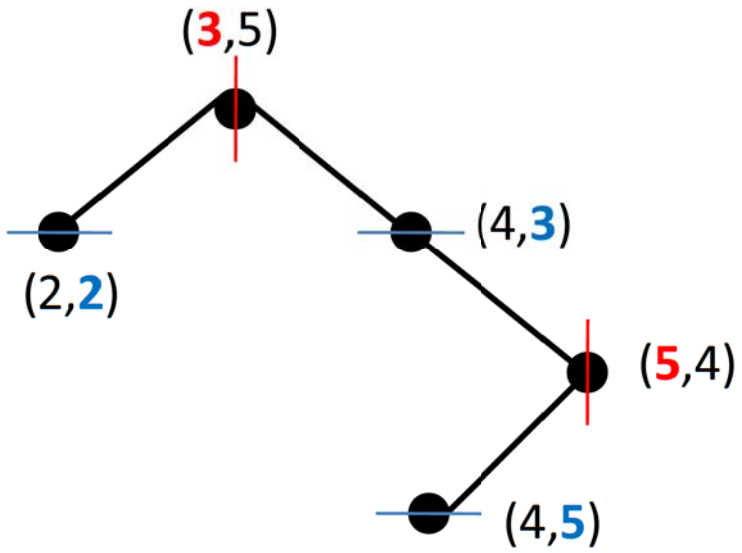




Example 2D-Tree Construction

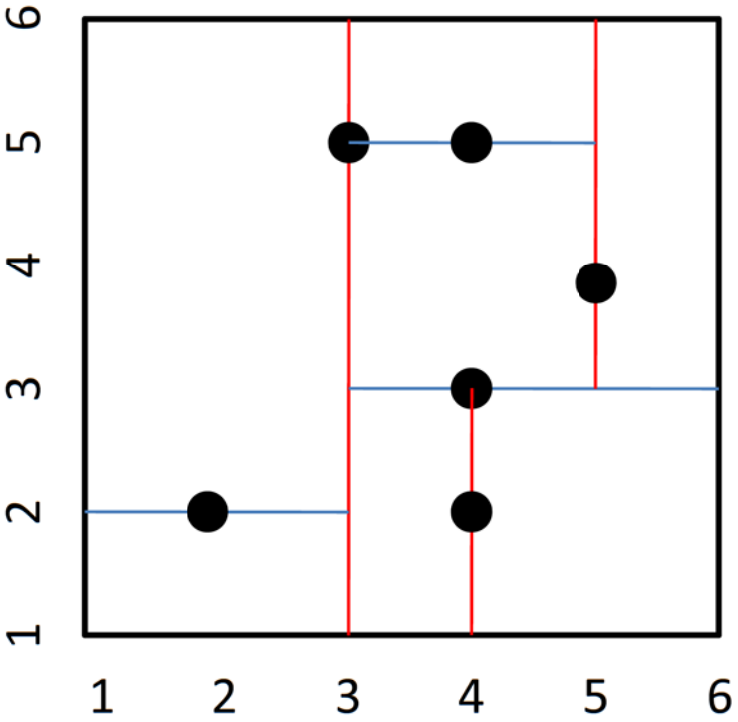
Input Sequence: (3,5) (4,3) (5,4) (4,5) (2,2) (4,2)

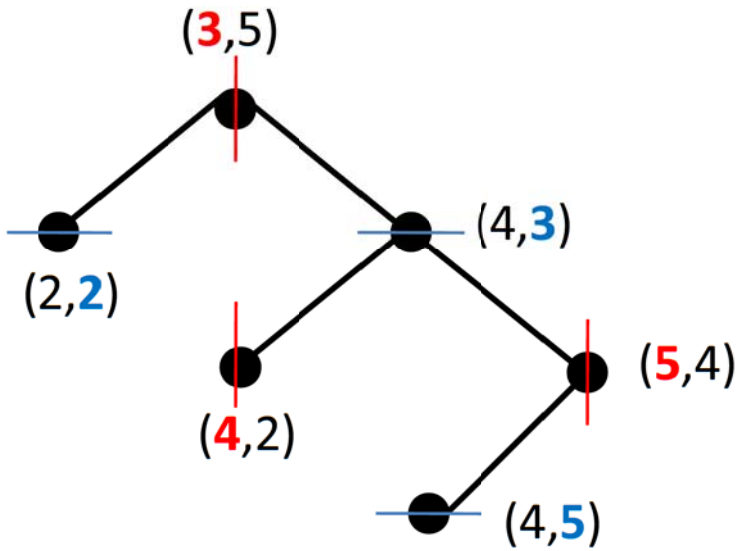




Example 2D-Tree Construction

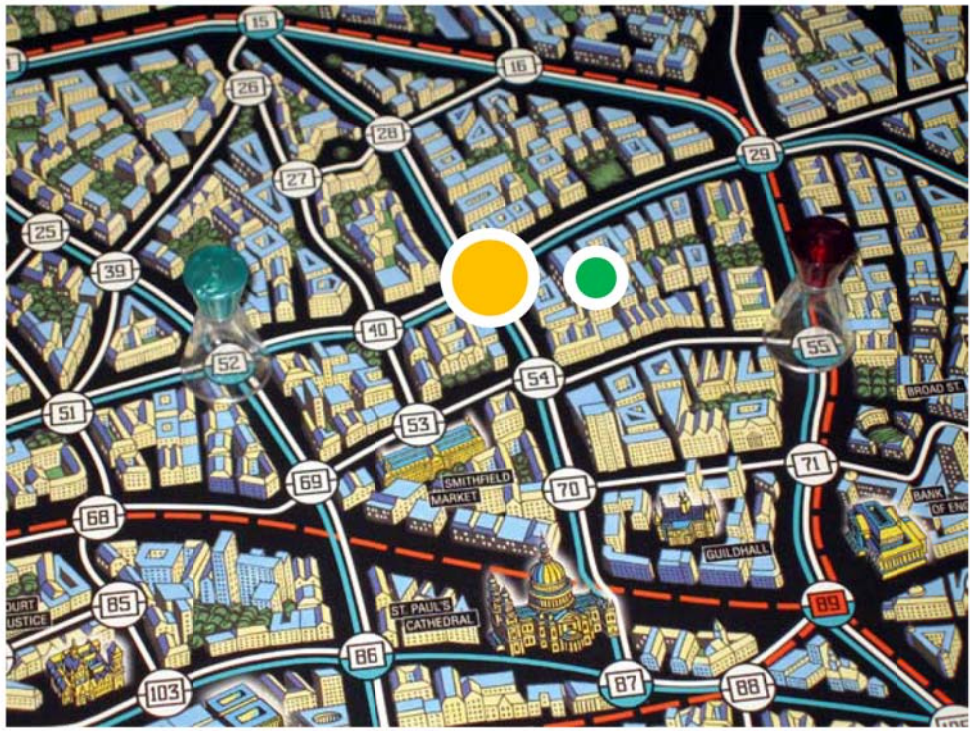
Input Sequence: (3,5) (4,3) (5,4) (4,5) (2,2) (4,2)





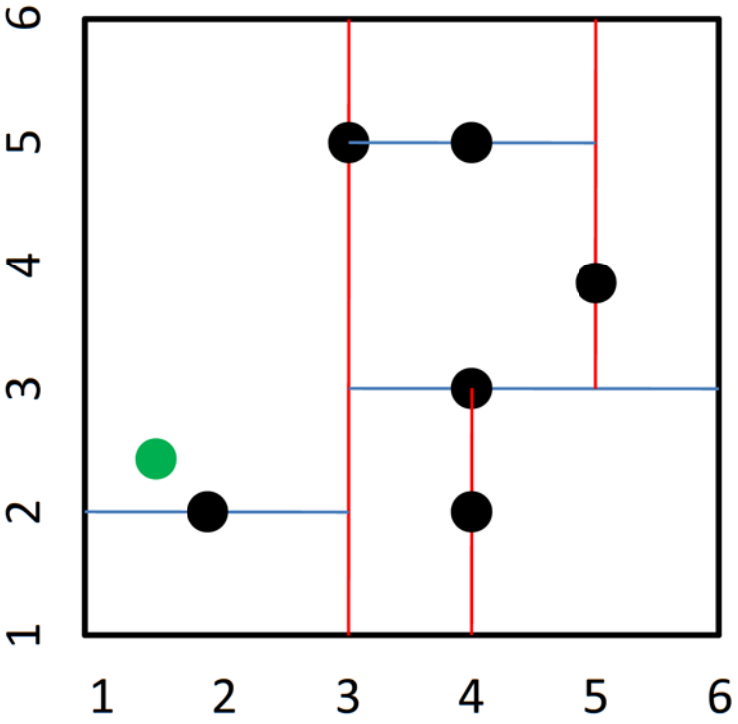
The Nearest Neighbour Problem

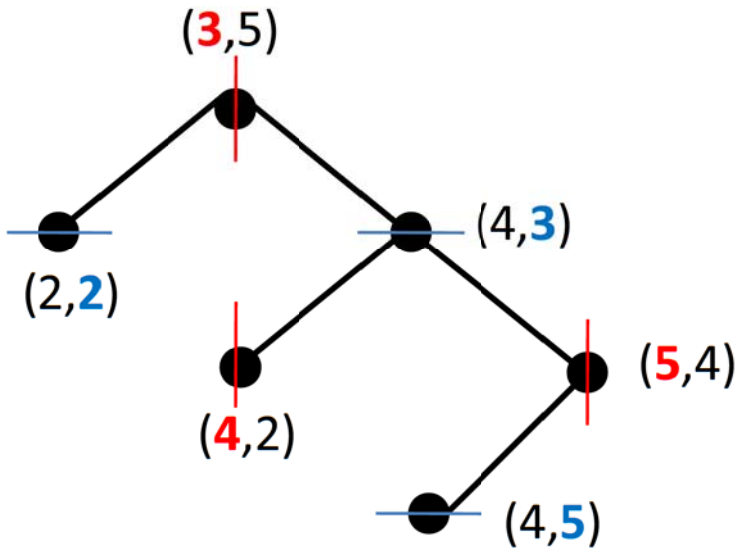
?



Nearest Neighbour Search

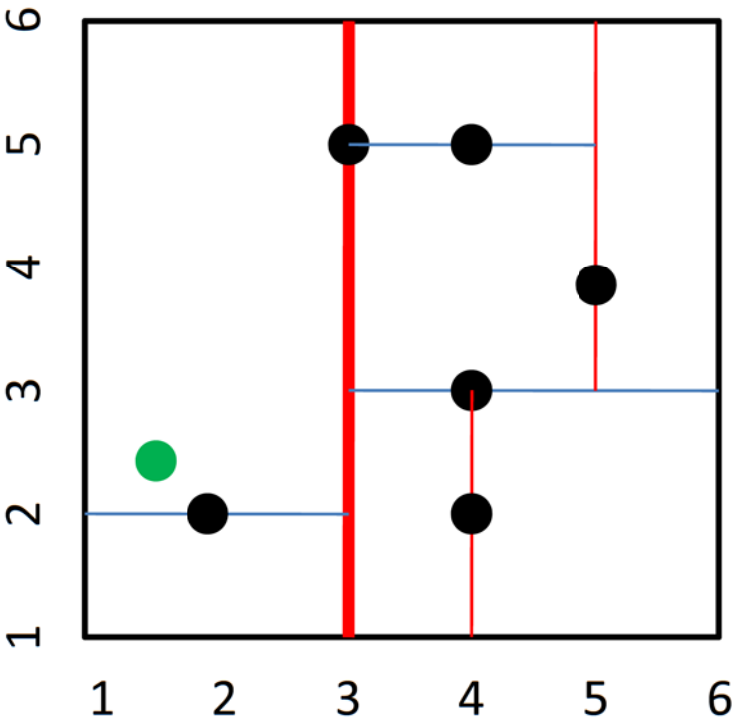
Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

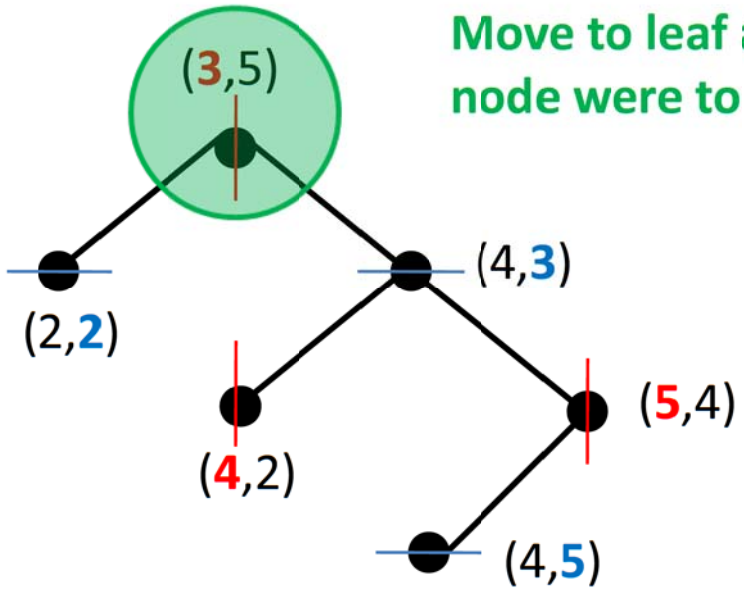




Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (1.5,2.5)**

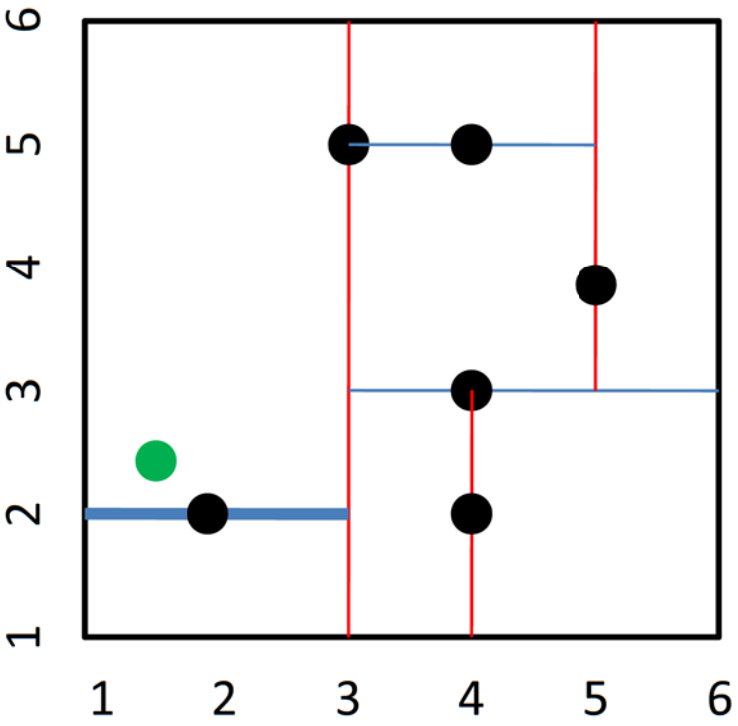


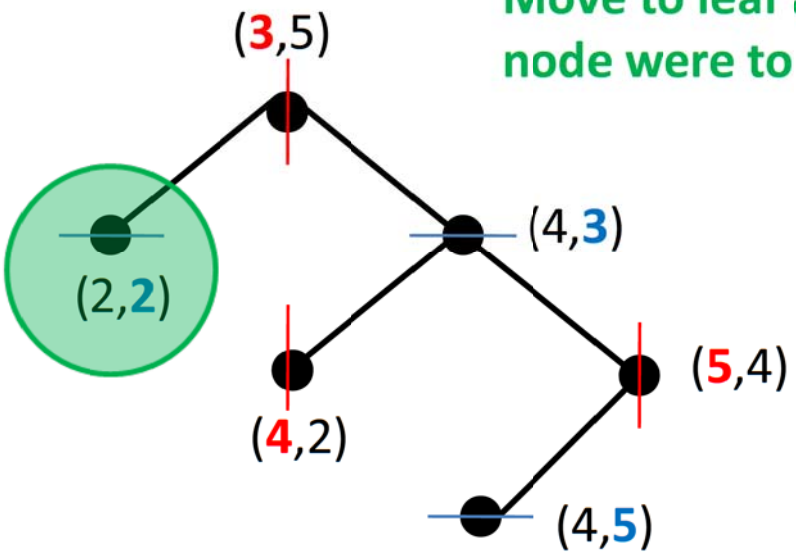


Move to leaf as if search node were to be added.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (1.5,2.5)**

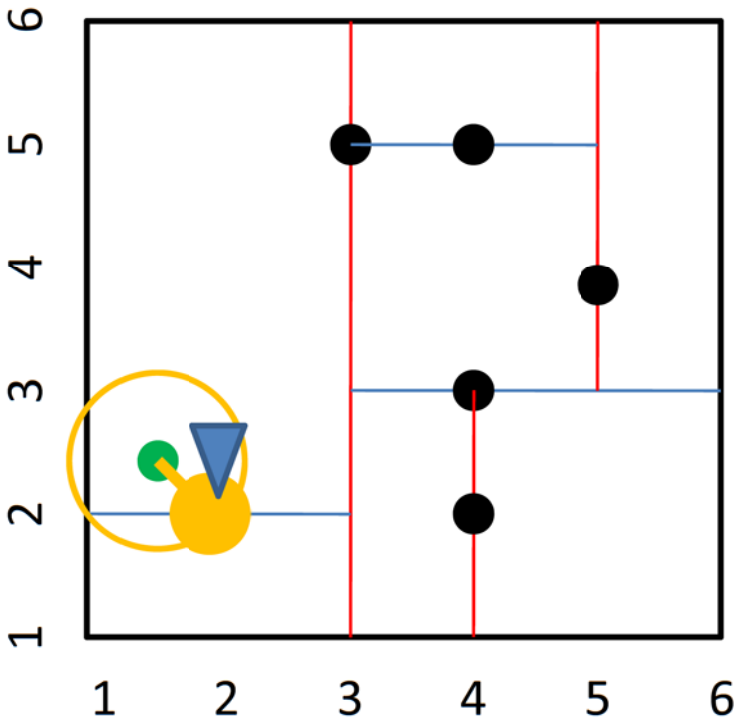




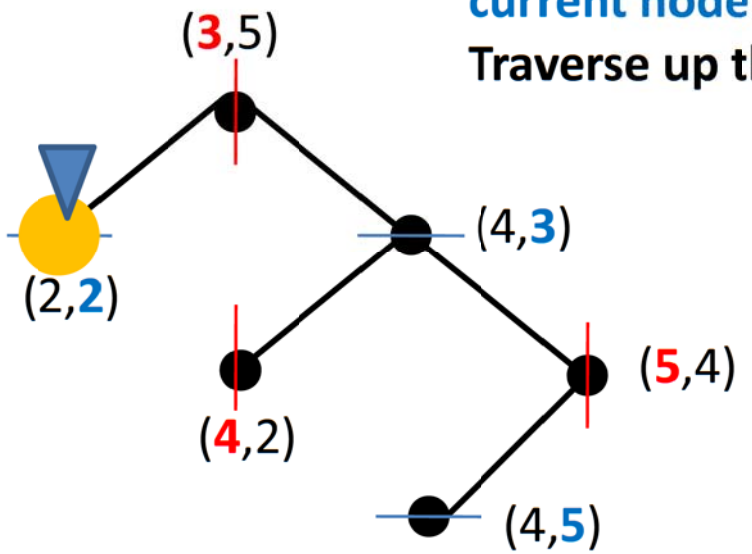
Move to leaf as if search node were to be added.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (1.5,2.5)**

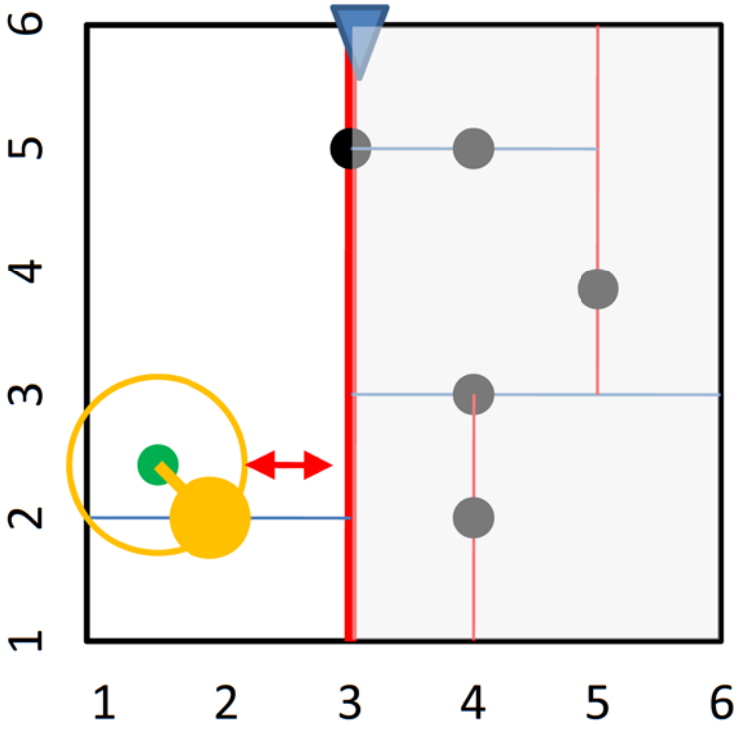


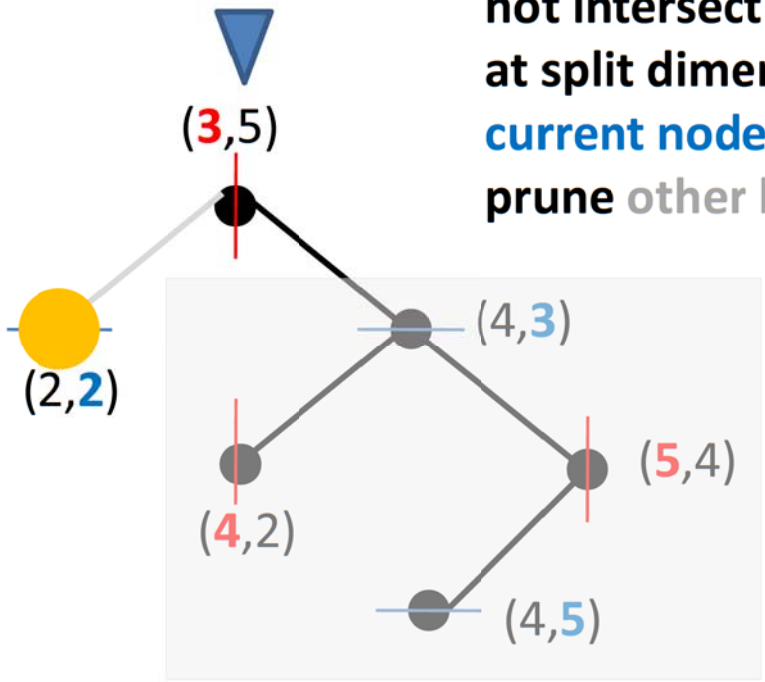
Make found leaf current best and current node. Traverse up the tree.



Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (1.5,2.5)**

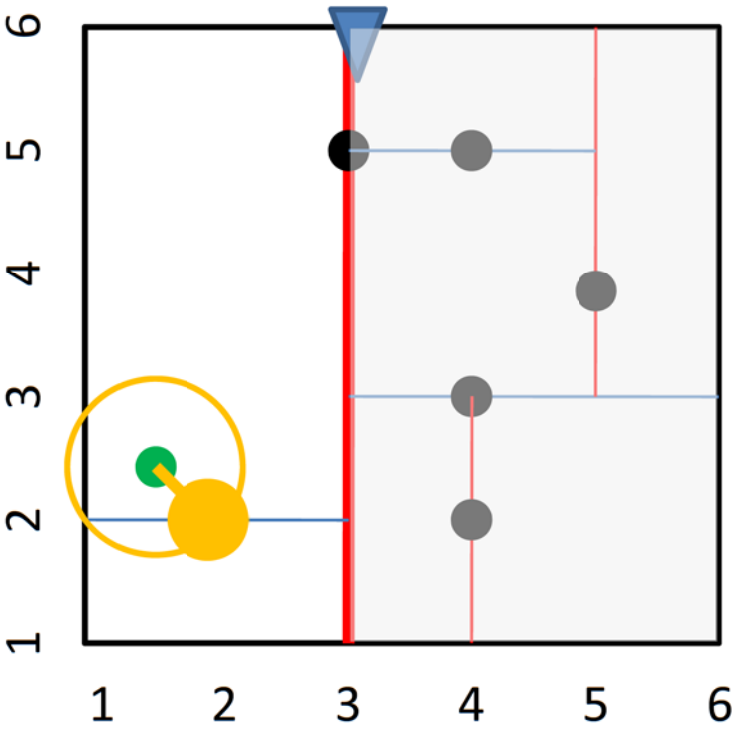


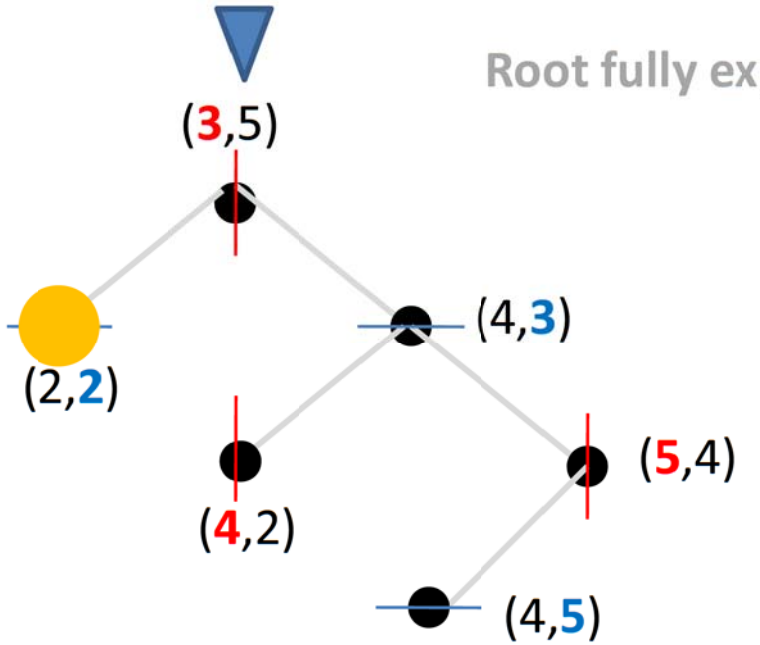


Hypersphere defined by current best does not intersect hyperplane at split dimension of current node, thus prune other branch.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (1.5,2.5)**



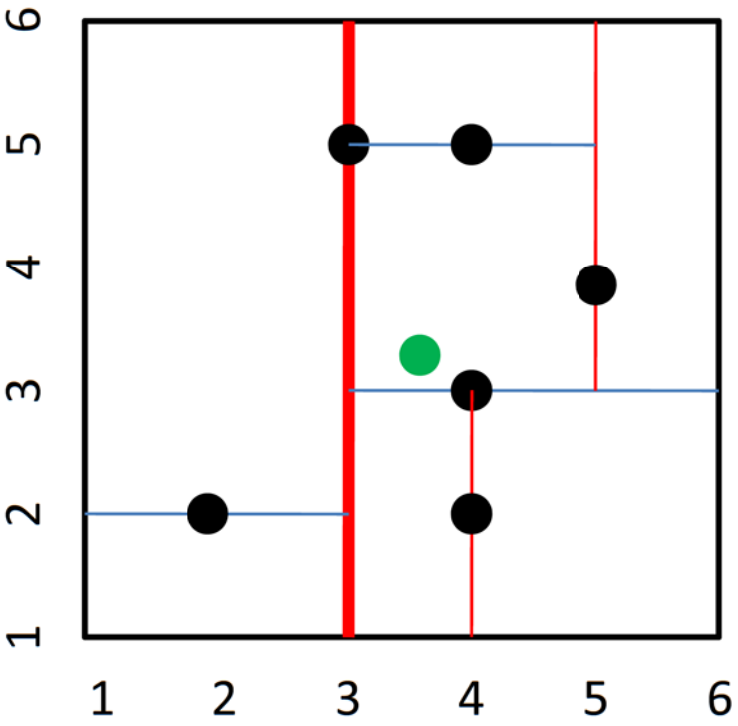


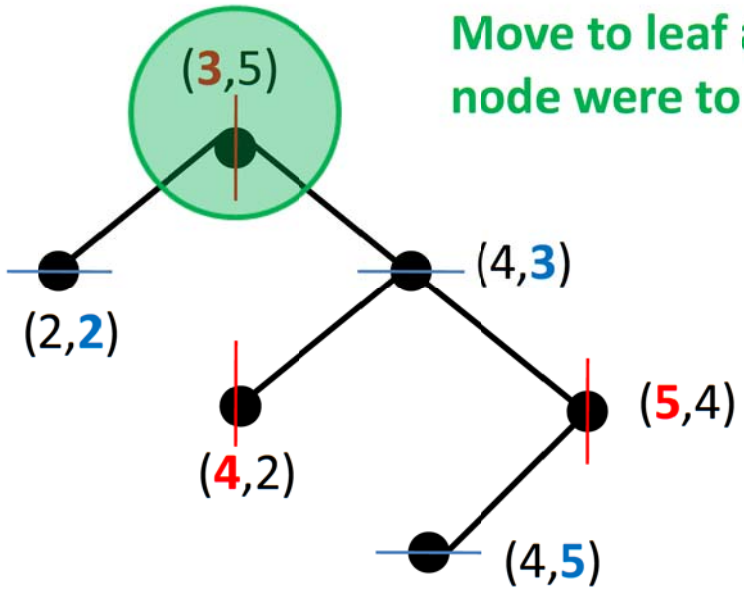
Root fully explored.

OUTPUT: (2,2)

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

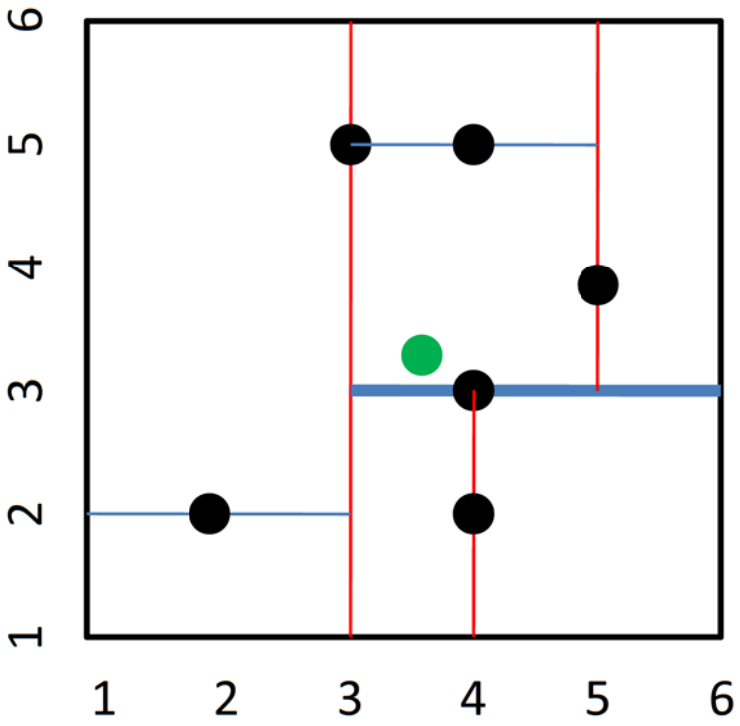


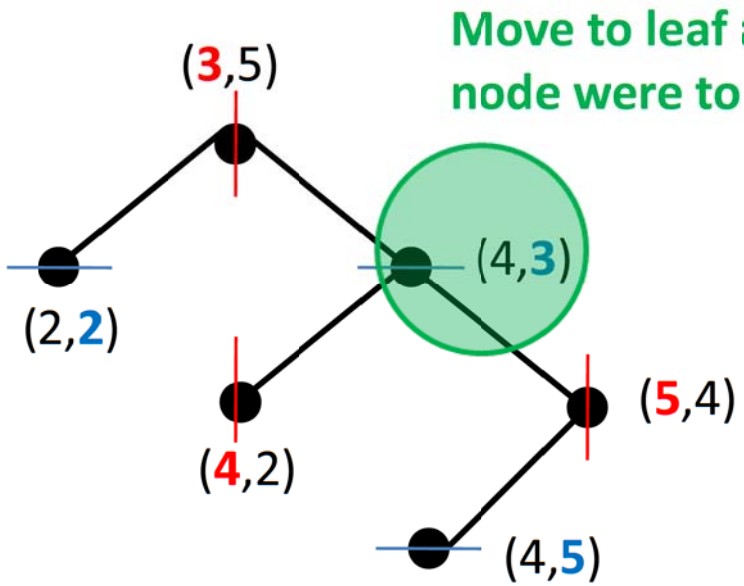


Move to leaf as if search node were to be added.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

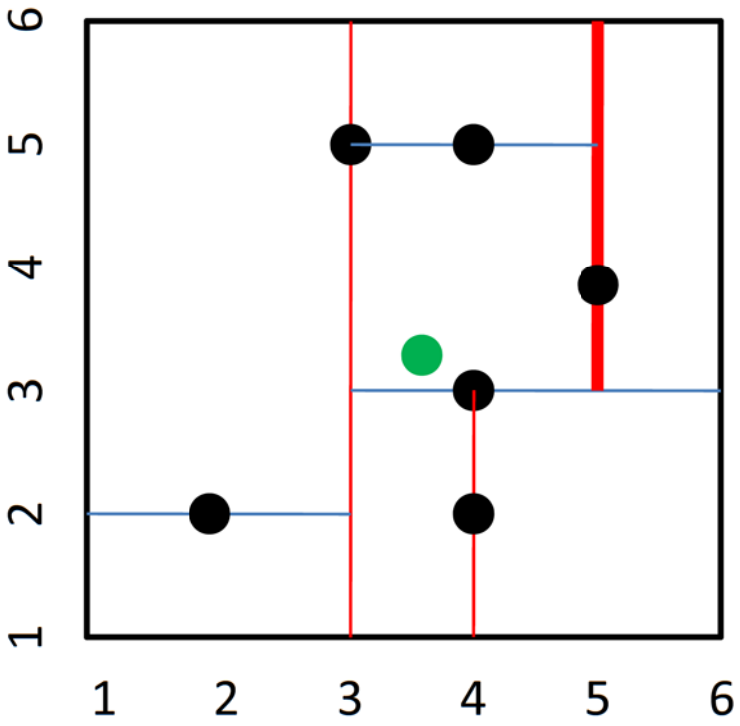


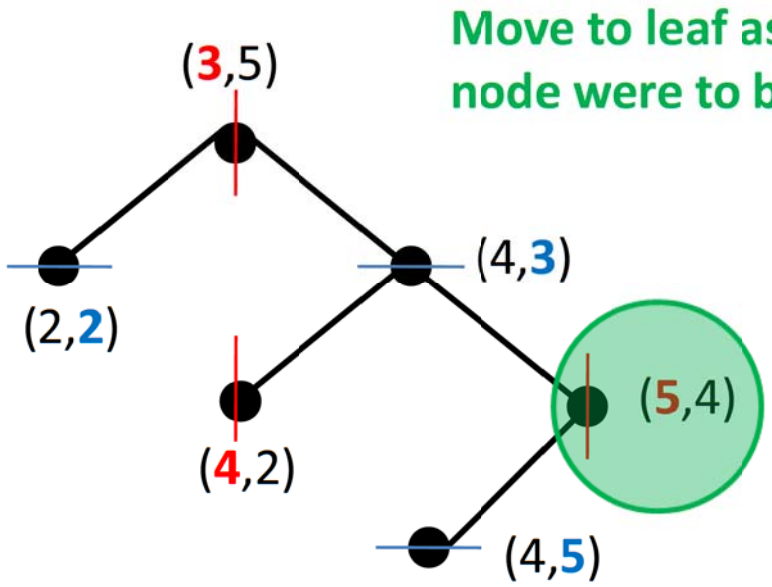


Move to leaf as if search node were to be added.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

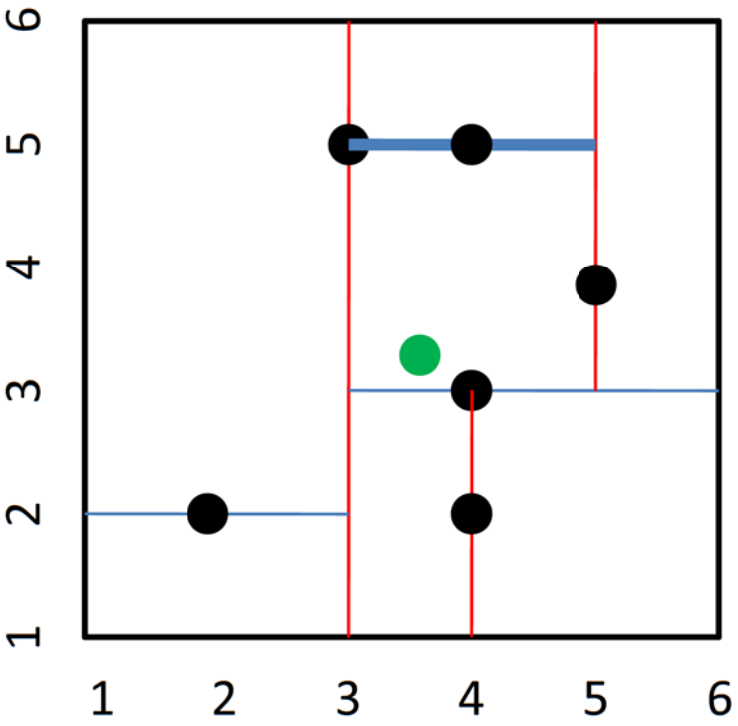


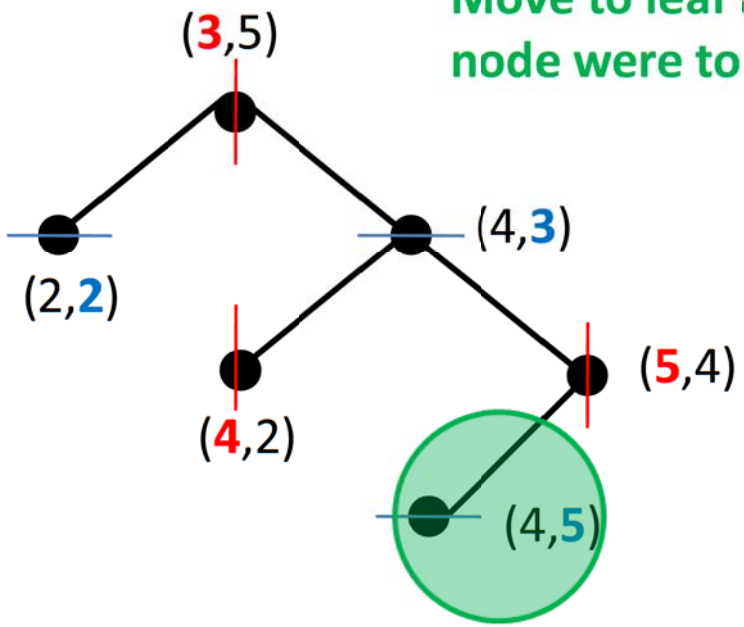


Move to leaf as if search node were to be added.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

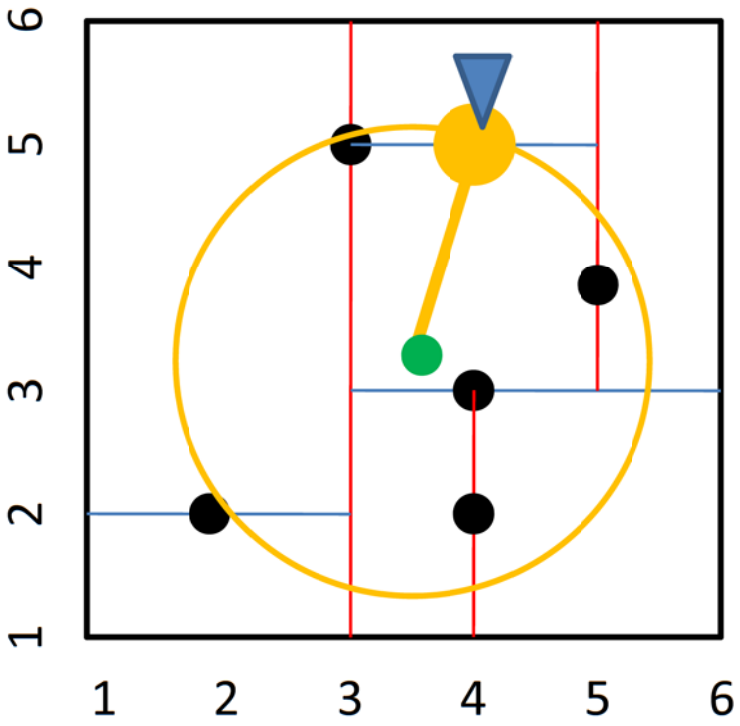




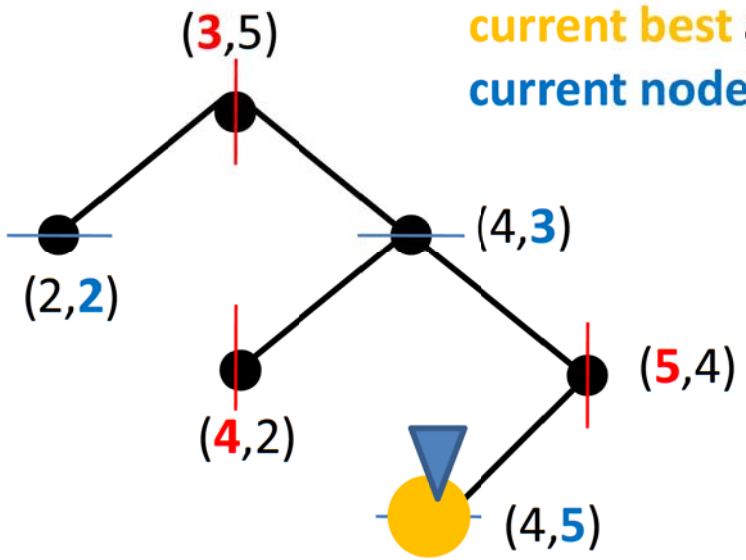
Move to leaf as if search node were to be added.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

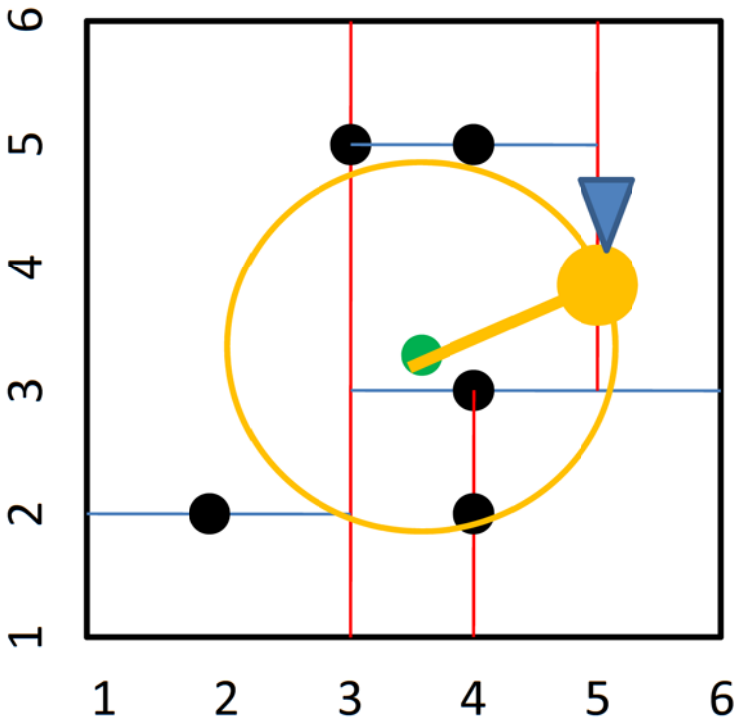


Make found leaf current best and current node.

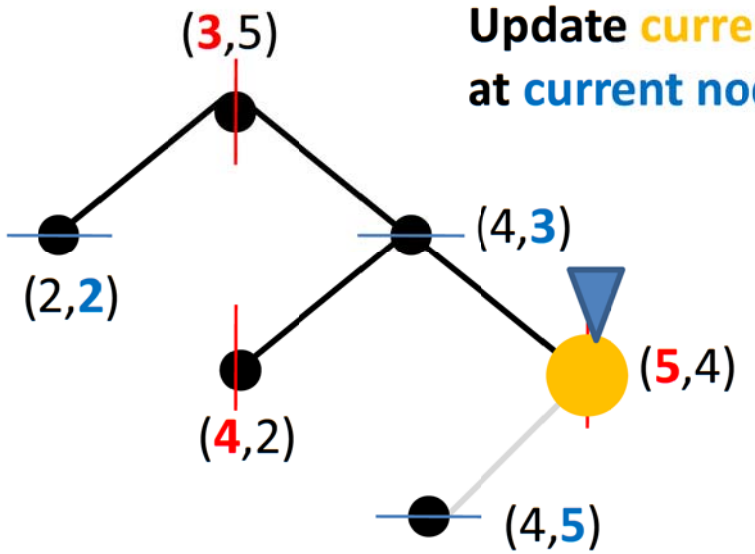


Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

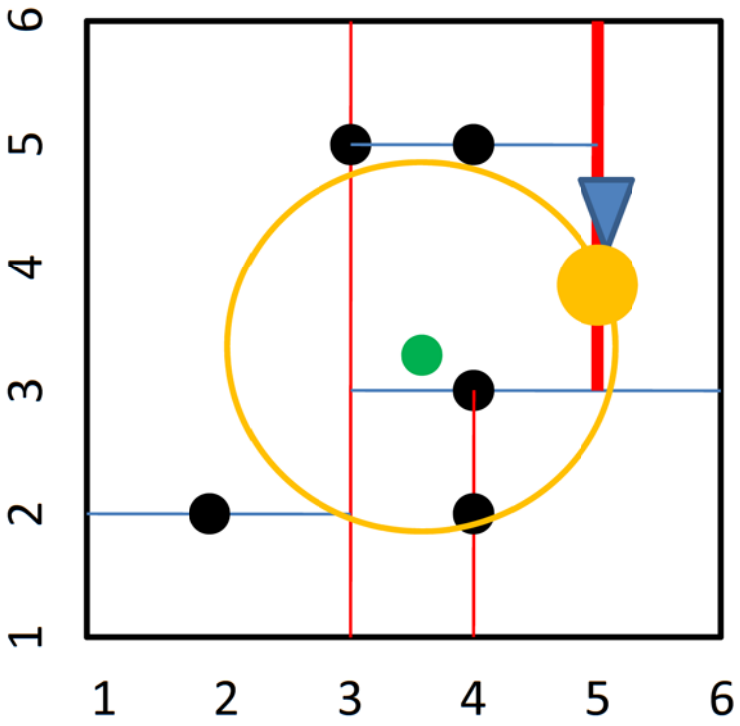


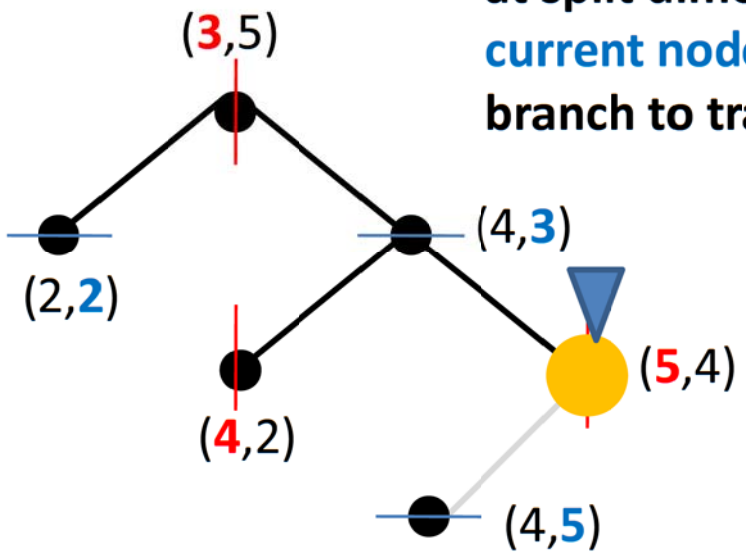
Traverse up the tree.
Update current best at current node.



Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

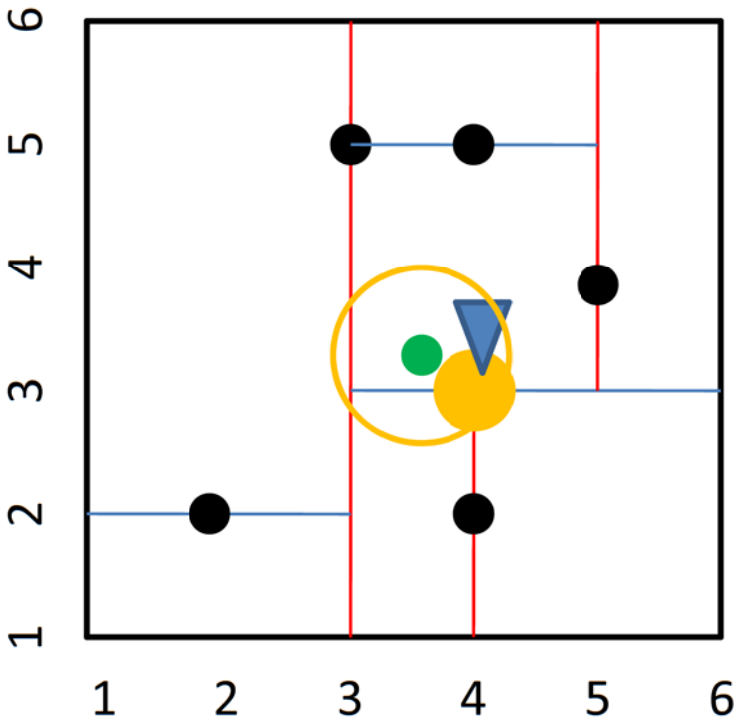




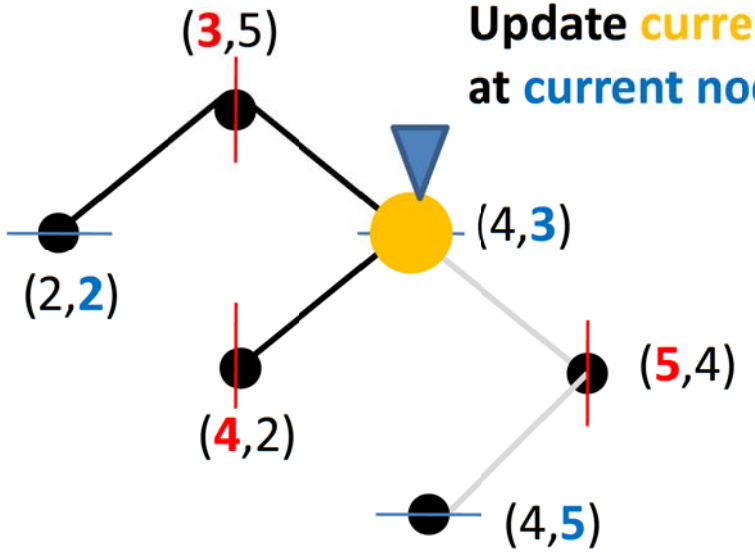
Hypersphere defined by current best DOES intersect hyperplane at split dimension of current node, but no branch to traverse down…

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

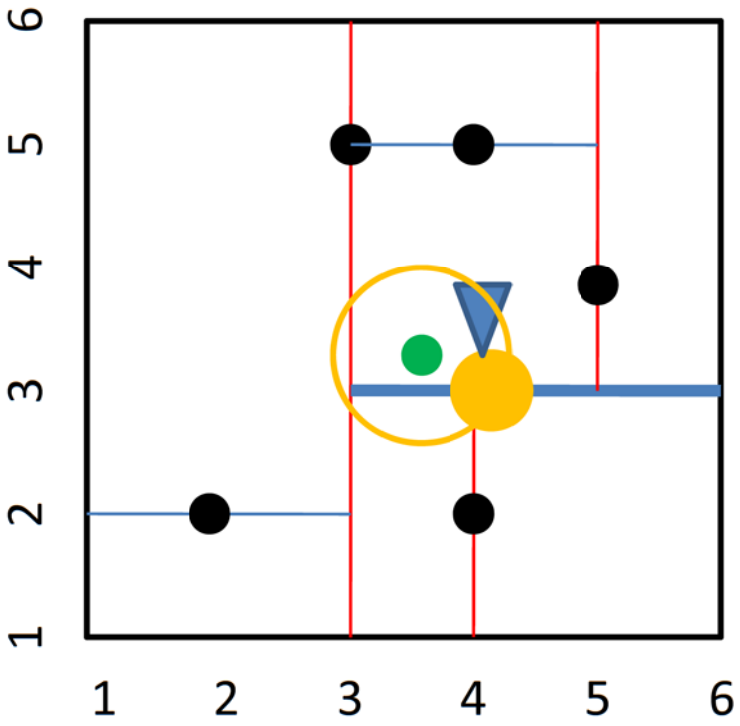


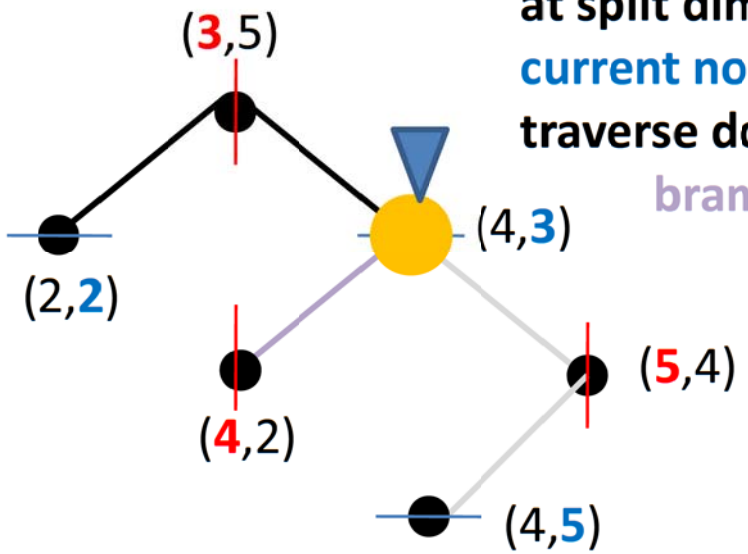
Traverse up the tree.
Update current best at current node.



Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

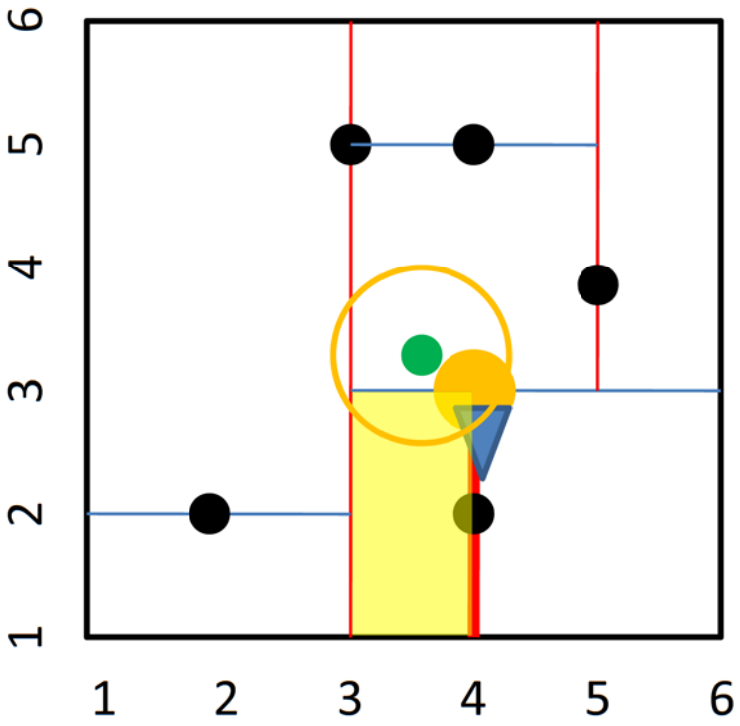


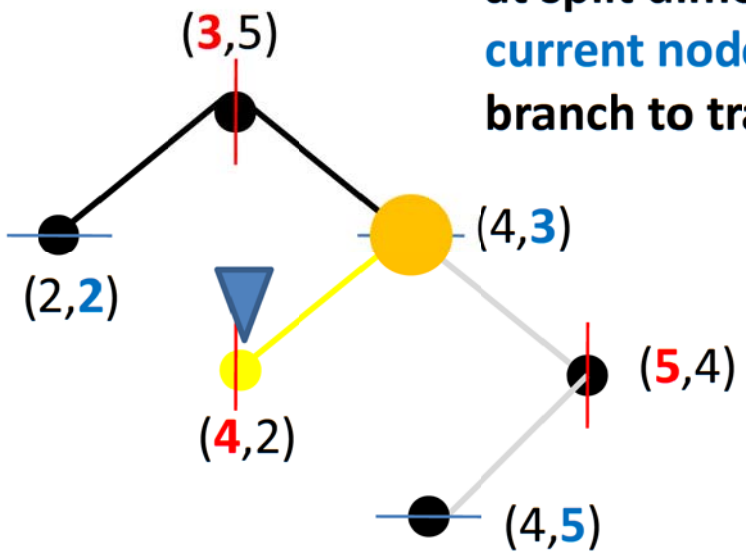


Hypersphere defined by current best DOES intersect hyperplane at split dimension of current node, thus traverse down alternate branch.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

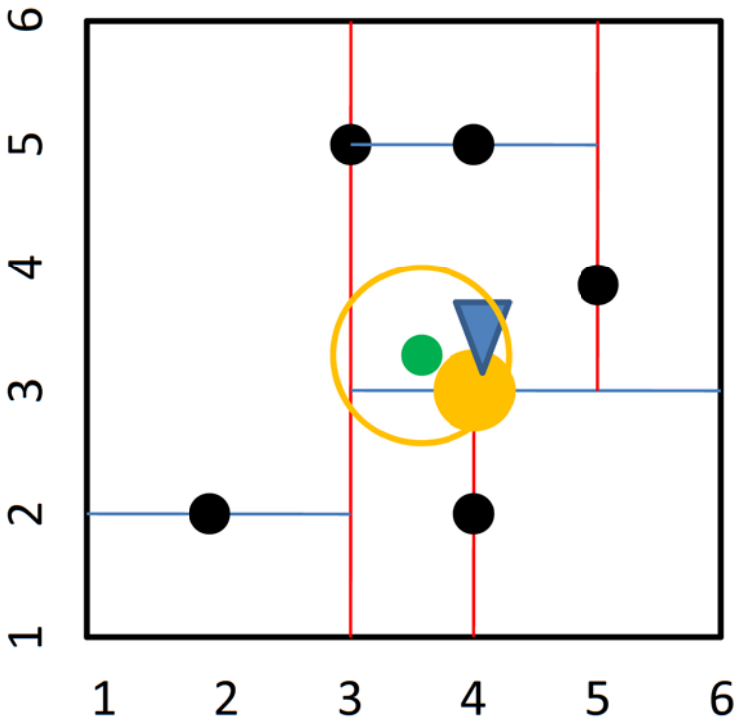




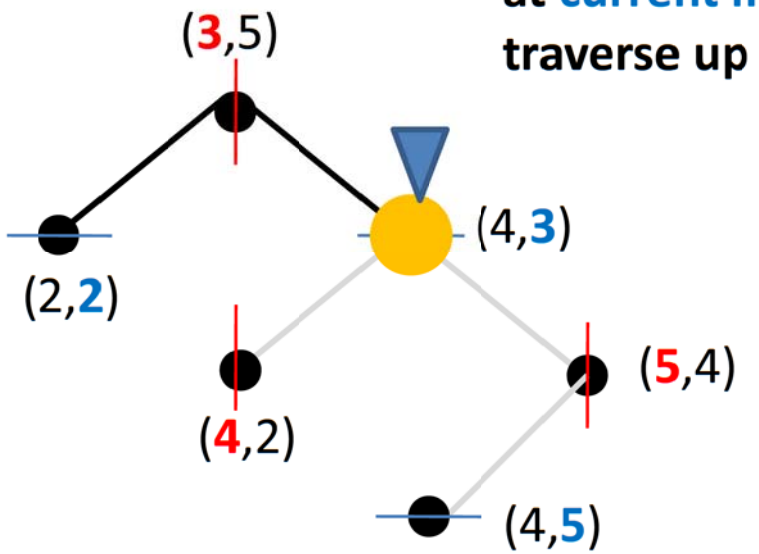
Hypersphere defined by current best DOES intersect hyperplane at split dimension of current node, but no branch to traverse down…

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

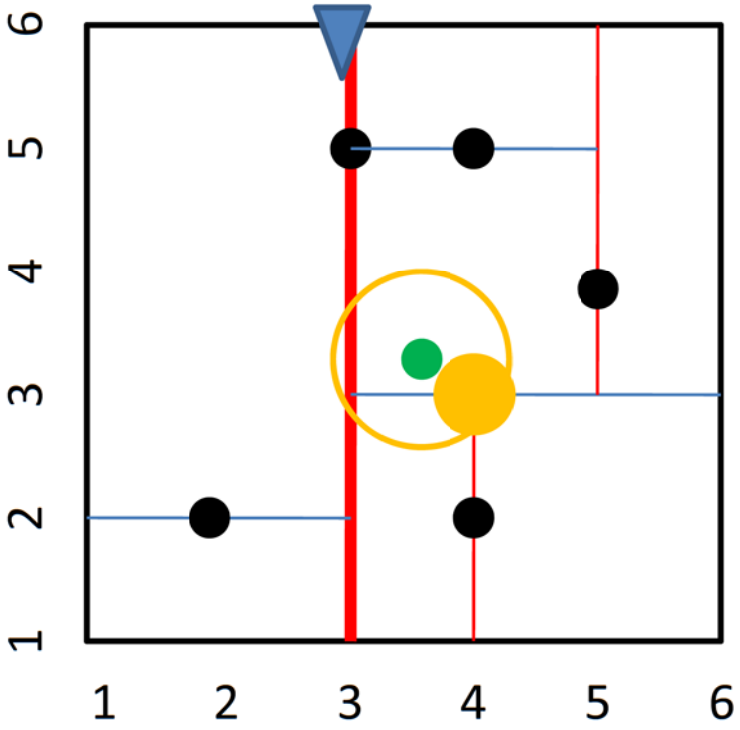


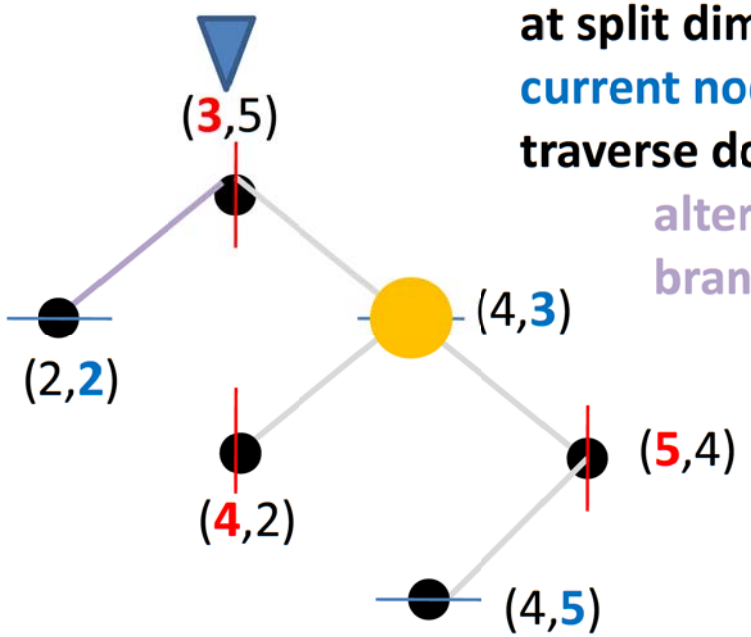
All branches tested at current node, traverse up the tree.



Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

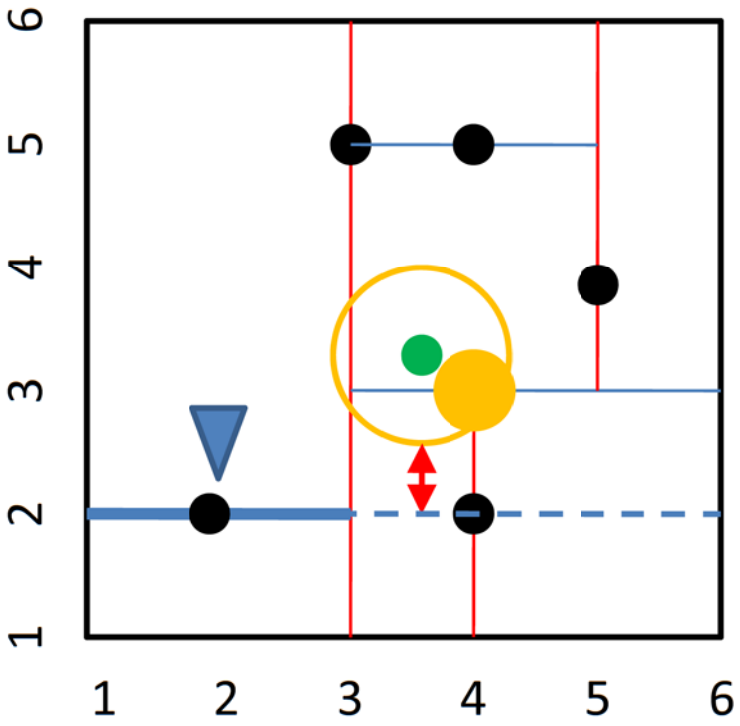


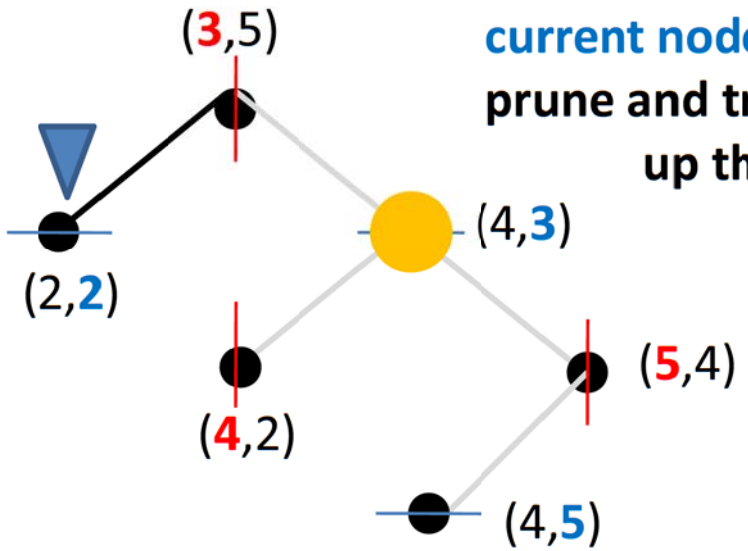


Hypersphere defined by current best DOES intersect hyperplane at split dimension of current node, thus traverse down alternate branch.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**

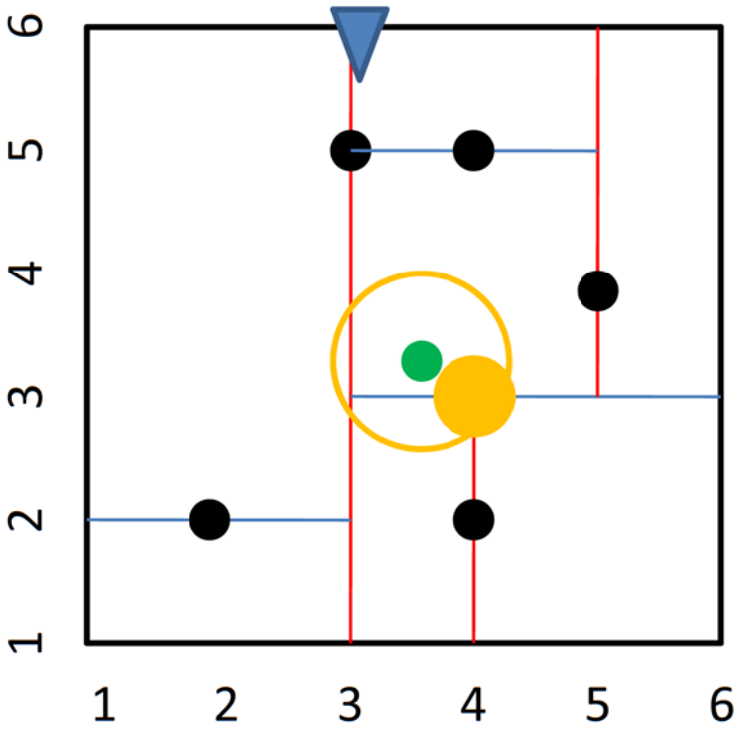


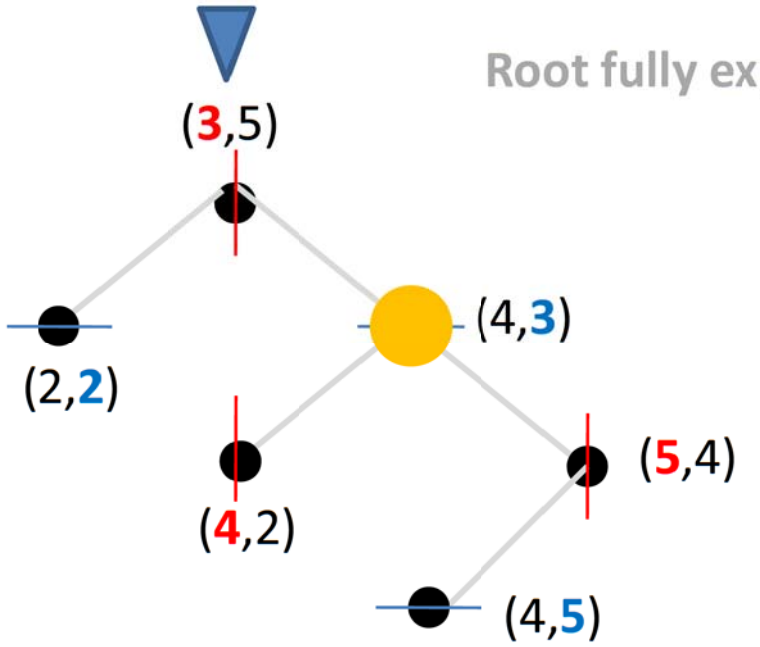


Hypersphere defined by current best does not intersect hyperplane at split dimension of current node, thus prune and traverse up the tree.

Nearest Neighbour Search

Task: Find Nearest Neighbour of **Search Node (3.5,3.3)**





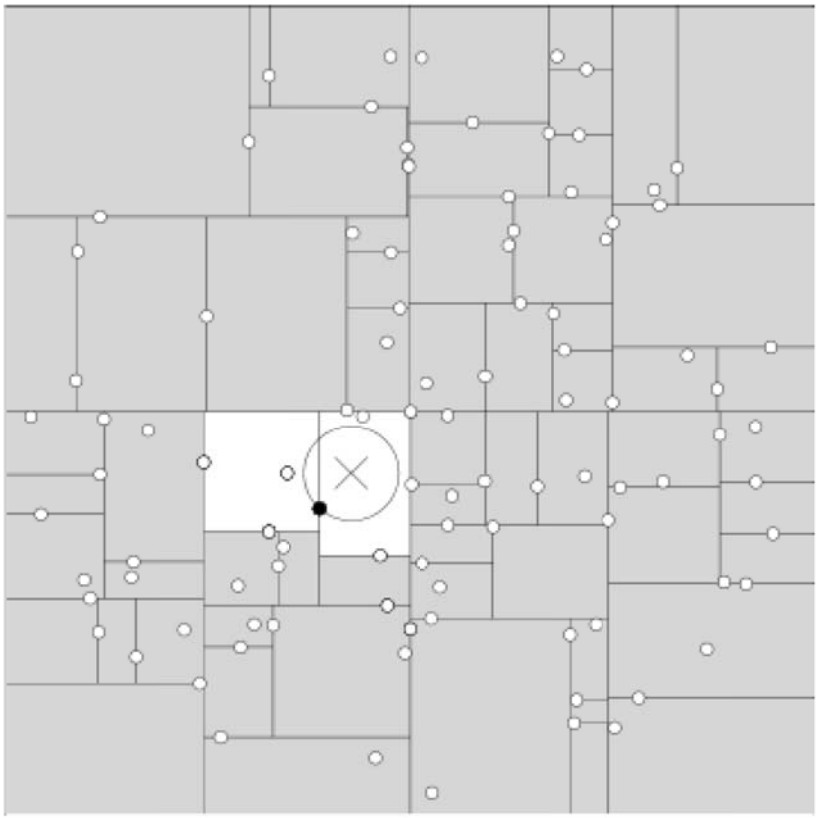
Root fully explored.

OUTPUT: (4,3)

Nearest Neighbour Algorithm

1) Starting with the root node, move down a path of the kD-Tree in the same way that one would if the search point were being inserted. Once a leaf node is reached, save the leaf as both the current best and current node
2) Exit with current best if root node is current node and fully tested
3) If the current node is closer than the current best, then it becomes the current best
4) Check if the difference between the splitting coordinate of the search point and current node is less than the Euclidean distance from the search point to current best:
   - If YES and a branch of the current node is untested, move down this branch and make the reached node current node
   - Otherwise make the parent node the current node and label the branch as tested
5) Goto 2

Performance: Worst Case O(n), typical application case O(log n)



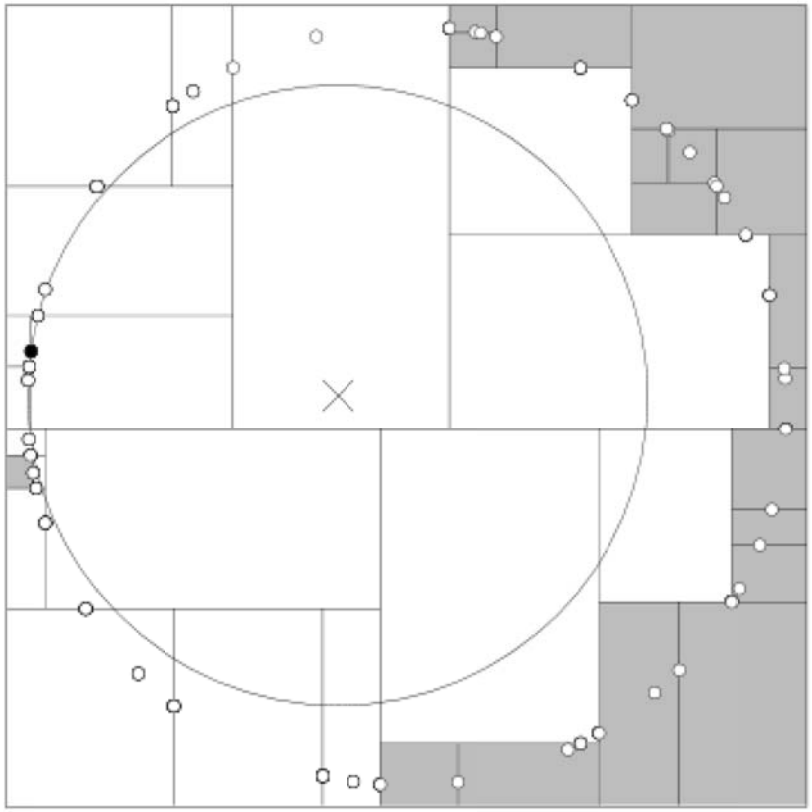


Image Source: A.W. Moore (PhD Thesis, Cambridge 1991)