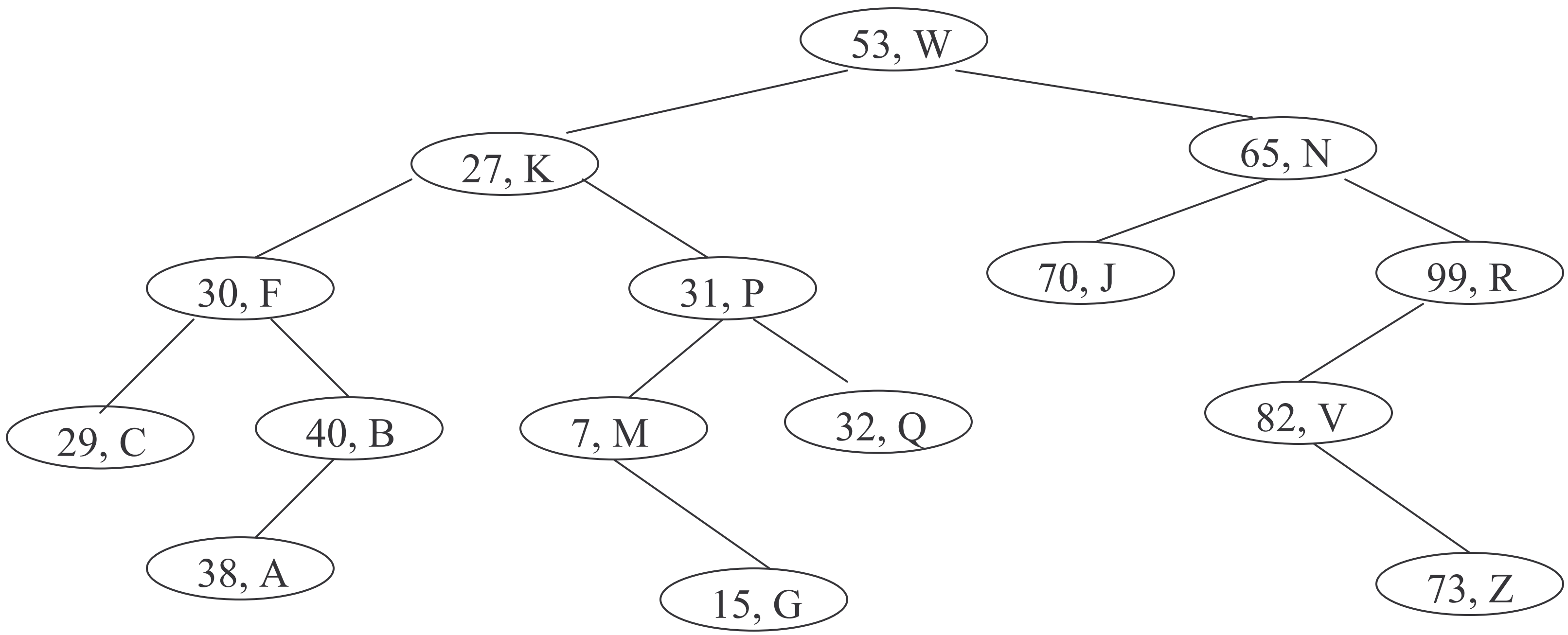
CMSC 341

K-D Trees

K-D Tree

- Introduction
  - Multiple dimensional data
    - Range queries in databases of multiple keys:
      Ex. find persons with
      $34 \le age \le 49$ and $\$100k \le annual\ income \le \$150k$
    - GIS (geographic information system)
    - Computer graphics
  - Extending BST from one dimensional to k-dimensional
    - It is a binary tree
    - Organized by levels (root is at level 0, its children level 1, etc.)
    - Tree branching at level 0 according to the first key, at level 1 according to the second key, etc.
- KdNode
  - Each node has a vector of keys, in addition to the two pointers to its left and right subtrees.

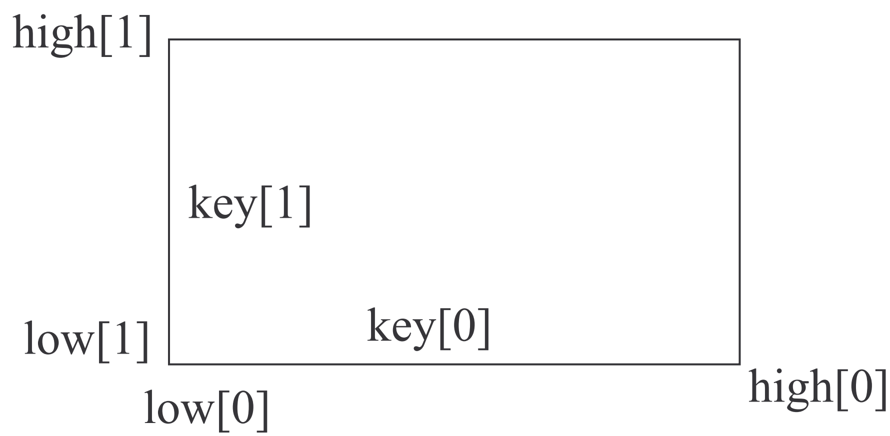
K-D Tree



A 2-D tree example

K-D Tree Operations

- Insert
  - A 2-D item (vector of size 2 for the two keys) is inserted
  - New node is inserted as a leaf
  - Different keys are compared at different levels
- Find/print with an orthogonal (square) range



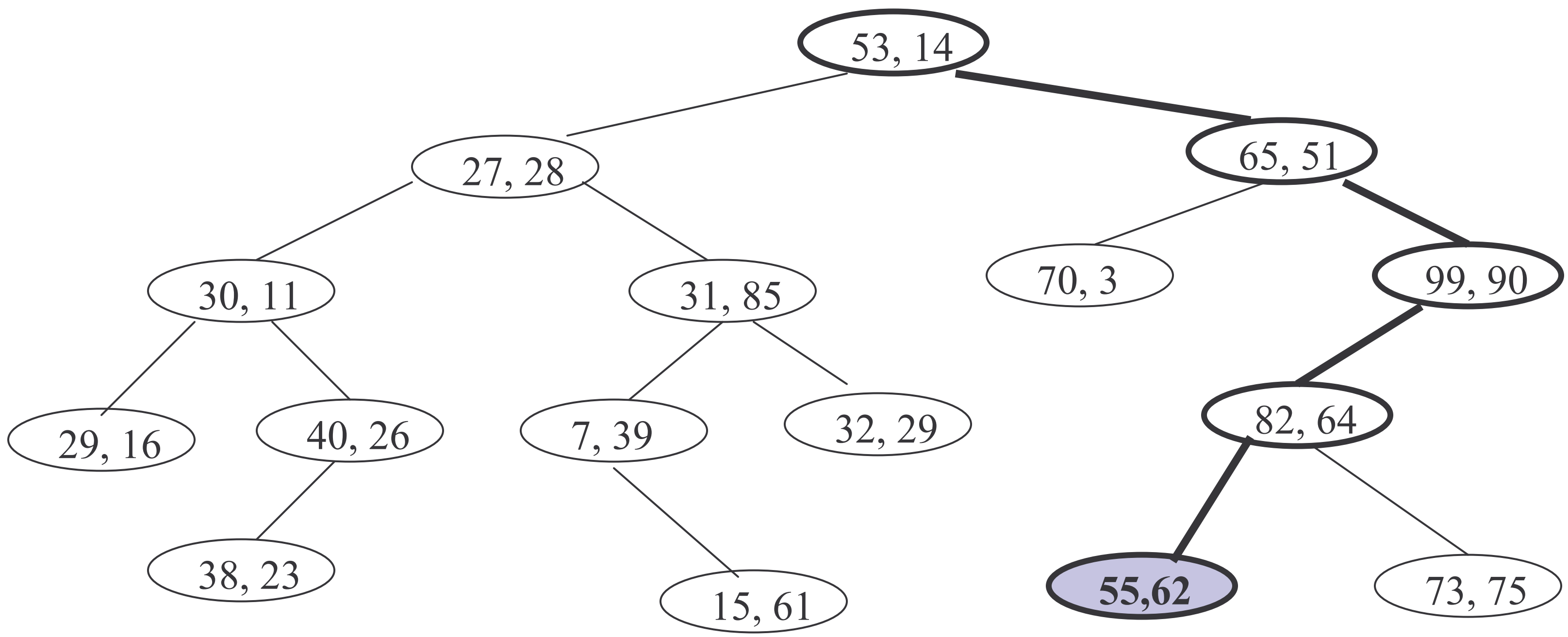
  - exact match: insert (low[level] = high[level] for all levels)
  - partial match: (query ranges are given to only some of the k keys, other keys can be thought in range $\pm \infty$)

K-D Tree Insertion

```
template <class Comparable>
void KdTree <Comparable>::insert(const vector<Comparable> &x)
{
   insert( x, root, 0);
}


template <class Comparable>
void KdTree <Comparable>::
insert(const vector<Comparable> &x, KdNode * & t, int level)
{
   if (t == NULL)
      t = new KdNode(x);
   else if (x[level] < t->data[level])
      insert(x, t->left, 1 - level);
   else
      insert(x, t->right, 1 - level);
}
```

Insert (55, 62) into the following 2-D tree



K-D Tree: PrintRange

```
/**
 * Print items satisfying
 * low[0] <= x[0] <= high[0] and
 * low[1] <= x[1] <= high[1]
 */

template <class Comparable>
void KdTree <Comparable>::
PrintRange(const vector<Comparable> &low,
           const vector<Comparable> & high) const
{
  PrintRange(low, high, root, 0);
}
```

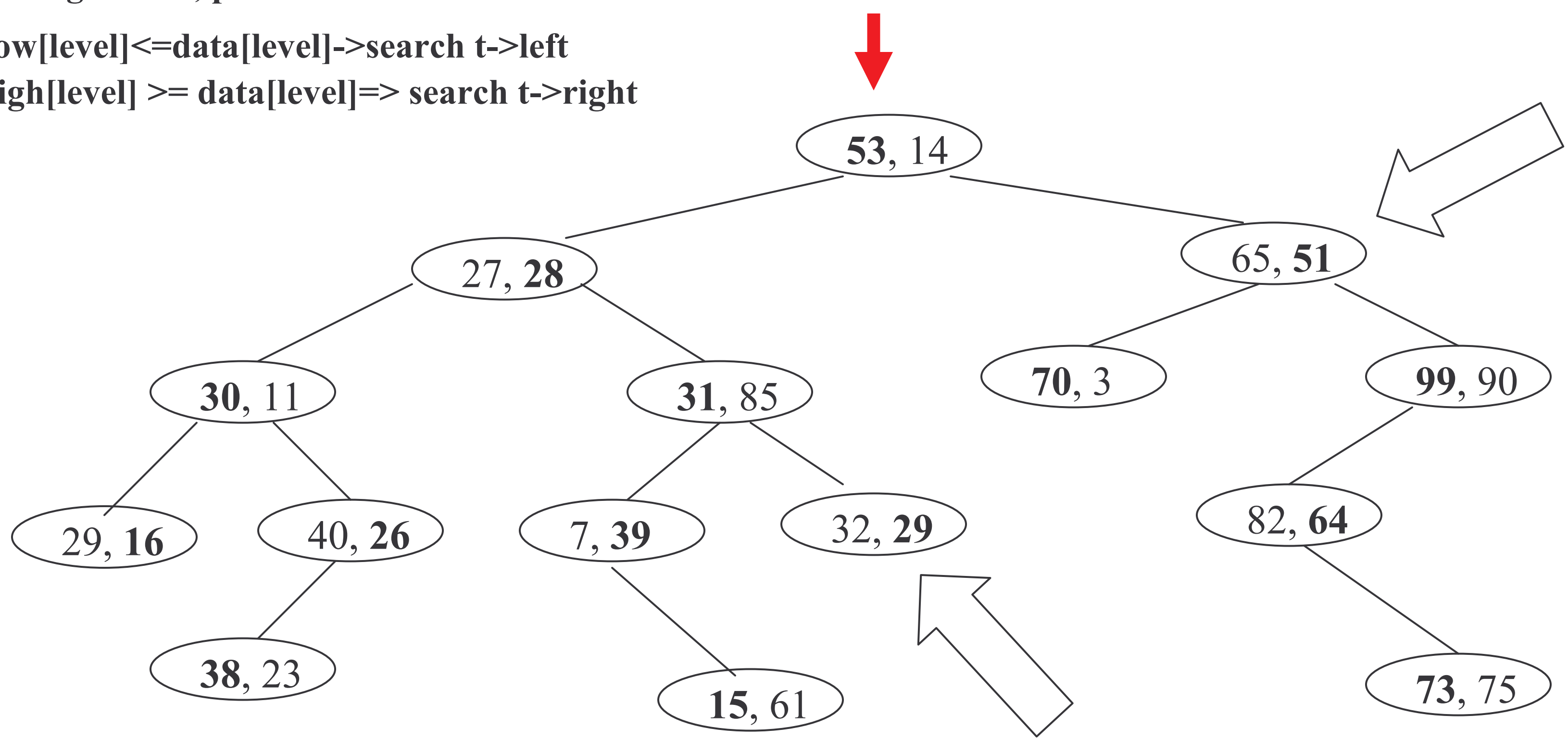
K-D Tree: PrintRange (cont’d)

```
template <class Comparable>
void KdTree <Comparable>::
PrintRange(const vector<Comparable> &low,
           const vector<Comparable> &high,
           KdNode * t, int level)
{
   if (t != NULL)
   {
     if ((low[0] <= t->data[0] && t->data[0] <= high[0])
     &&   (low[1] <= t->data[1] && t->data[1] <= high[1]))
         cout << “(” << t->data[0] ”,”
              << t->data[1] << “)” << endl;
     if (low[level] <= t->data[level])
         PrintRange(low, high, t->left, 1 – level);
     if (high[level] >= t->data[level])
         PrintRange(low, high, t->right, 1 – level);
   }
}
```

printRange in a 2-D Tree

In range? If so, print cell

Low[level]<=data[level]->search t->left
High[level] >= data[level]=> search t->right



low[0] = 35, high[0] = 40;
low[1] = 23, high[1] = 30;

This subtree is never searched

Searching is "preorder". Efficiency is obtained by "pruning" subtrees from the search.

K-D Tree Performance

- Insert
  - Average and balanced trees: $O(\lg N)$
  - Worst case: $O(N)$
- Print/search with a square range query
  - Exact match: same as insert (low[level] = high[level] for all levels)
  - Range query: for M matches
    - Perfectly balanced tree:

      K-D trees: $O(M + kN^{(1-1/k)})$

      2-D trees: $O(M + \sqrt{N})$
    - Partial match

      in a random tree: $O(M + N^{\alpha})$ where $\alpha = (-3 + \sqrt{17}) / 2$

K-D Tree Performance

- More on range query in a perfectly balanced 2-D tree:
  - Consider one boundary of the square (say, low[0])
  - Let T(N) be the number of nodes to be looked at with respect to low[0]. For the current node, we may need to look at
    - One of the two children (e.g., node (27, 28), and
    - Two of the four grand children (e.g., nodes (30, 11) and (31, 85).
  - Write $T(N) = 2\ T(N/4) + c$, where N/4 is the size of subtrees 2 levels down (we are dealing with a perfectly balanced tree here), and $c = 3$.
  - Solving this recurrence equation:

    $T(N) = 2T(N/4) + c = 2(2T(N/16) + c) + c$

    …

    $= c(1 + 2 + \cdots + 2\char`^(\log_4 N) = 2\char`^(1+ \log_4 N) - 1$

    $= 2*2\char`^(\log_4 N) - 1 = 2\char`^ ((\log_2 N)/2) - 1 = O(\sqrt{N})$

K-D Tree Remarks

- Remove
  - No good remove algorithm beyond lazy deletion (mark the node as removed)
- Balancing K-D Tree
  - No known strategy to guarantee a balanced 2-D tree
  - Tree rotation does not work here
  - Periodic re-balance
- Extending 2-D tree algorithms to k-D
  - Cycle through the keys at each level