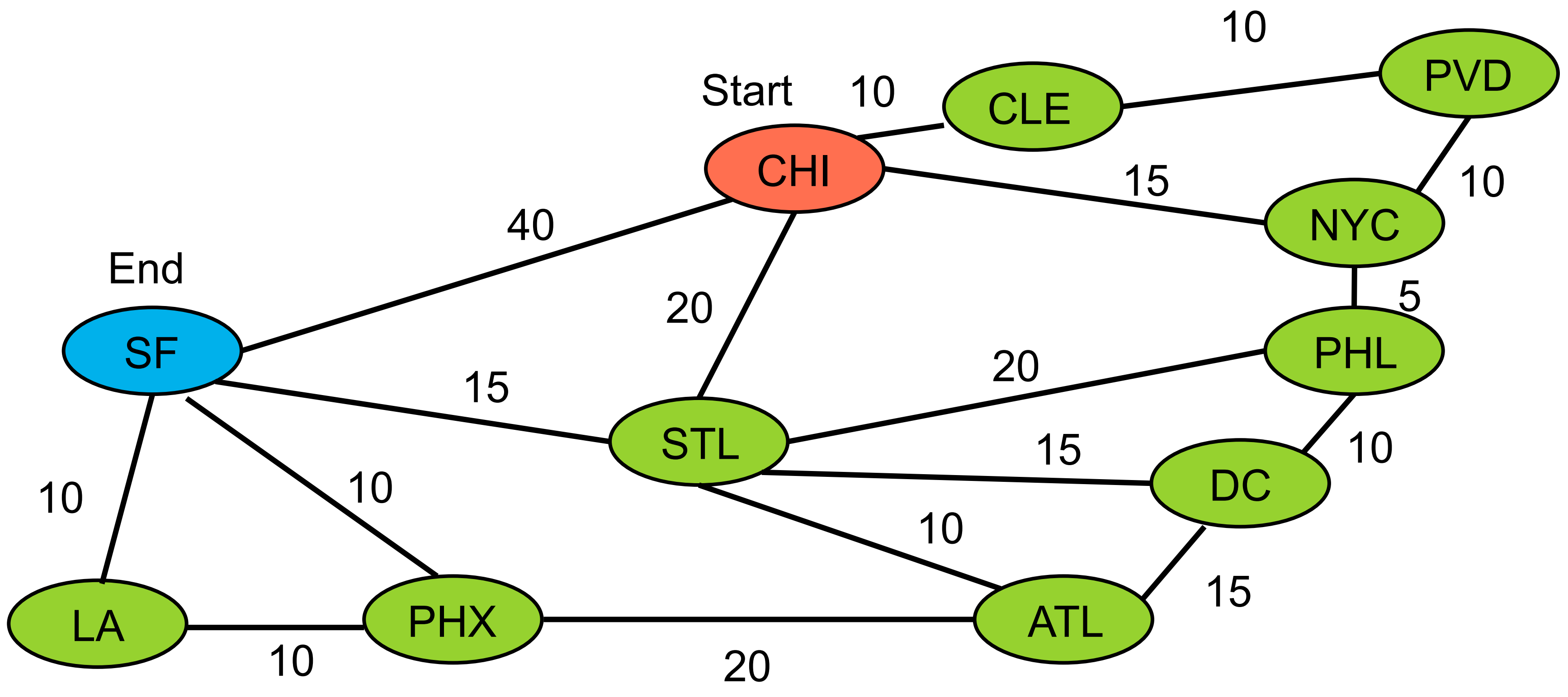
PRACTICAL IMPROVEMENT: A-STAR, KD-TREE, FIBONACCI HEAP

CS16: Introduction to Data Structures & Algorithms

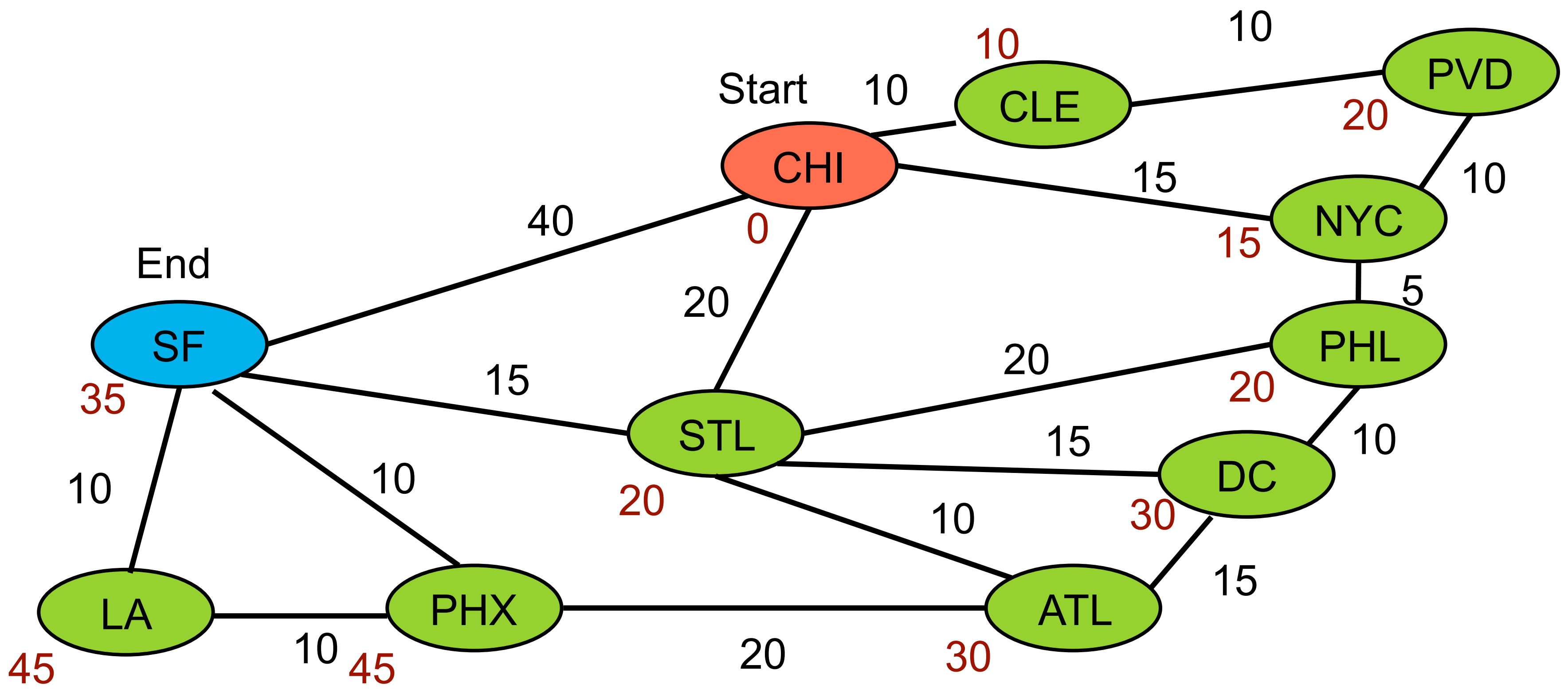
Outline

- A* Shortest-path Search
- KD-trees for Nearest Neighbor
- Fibonacci Heap

Road Trip

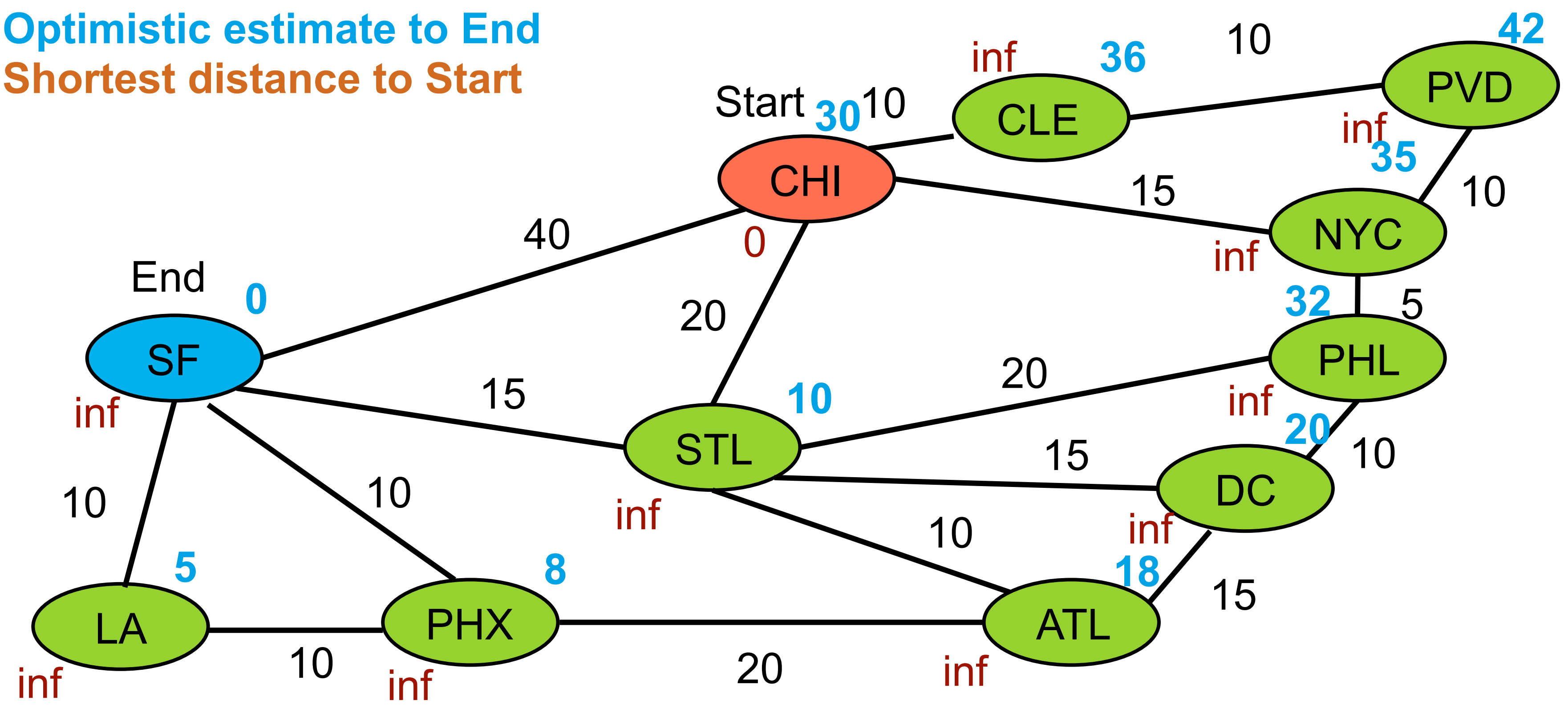

Start
CHI
End
SF
CLE
PVD
NYC
PHL
STL
DC
ATL
PHX
LA
10
10
15
10
40
5
20
20
15
15
10
10
10
10
15
10
20

Road Trip


Start
CHI
0
End
SF
35
CLE
10
PVD
20
NYC
15
PHL
20
STL
20
DC
30
ATL
30
PHX
45
LA
45
10
10
15
10
40
20
5
15
20
15
10
10
10
10
10
20
15
10

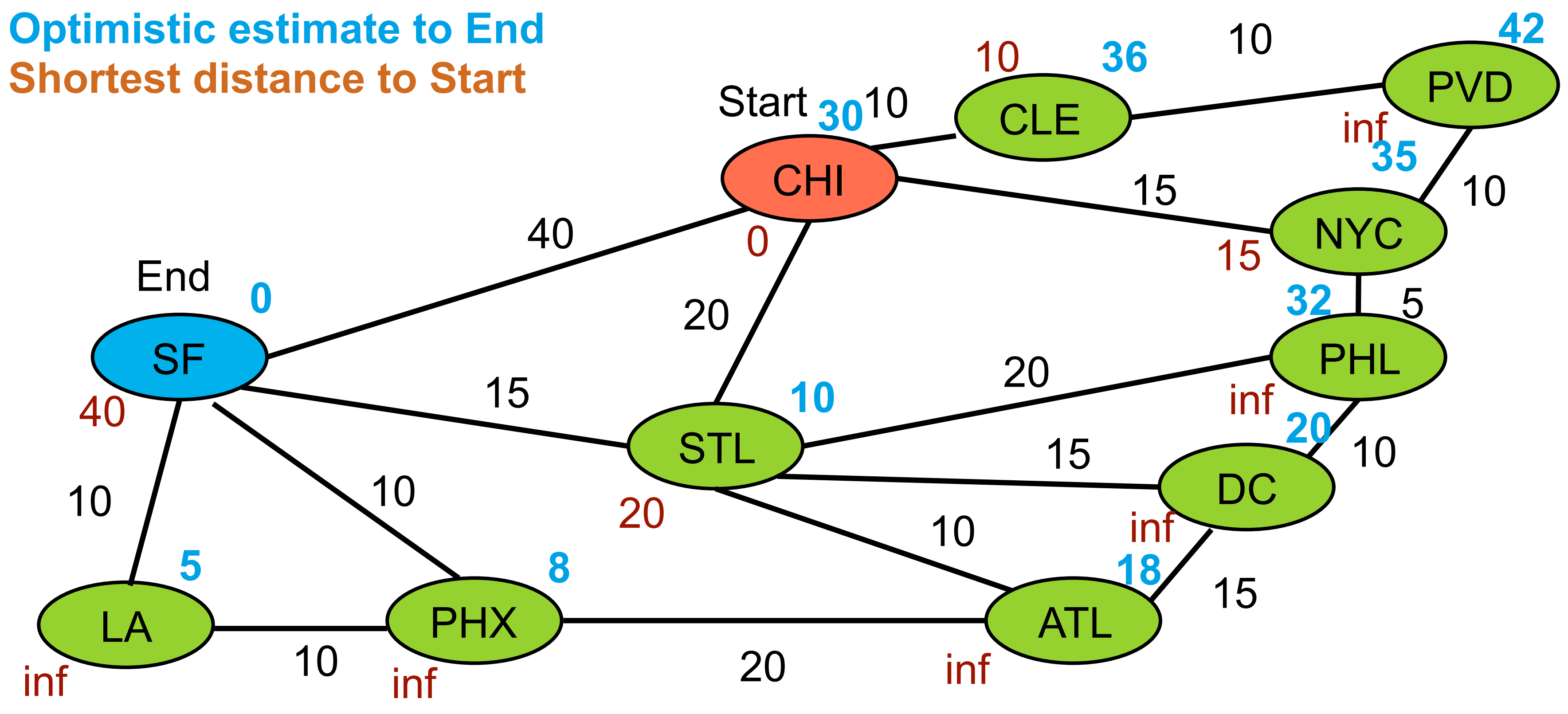
Optimistic Estimates

Optimistic estimate to End
Shortest distance to Start



PQ: (CHI,30) (…,inf)

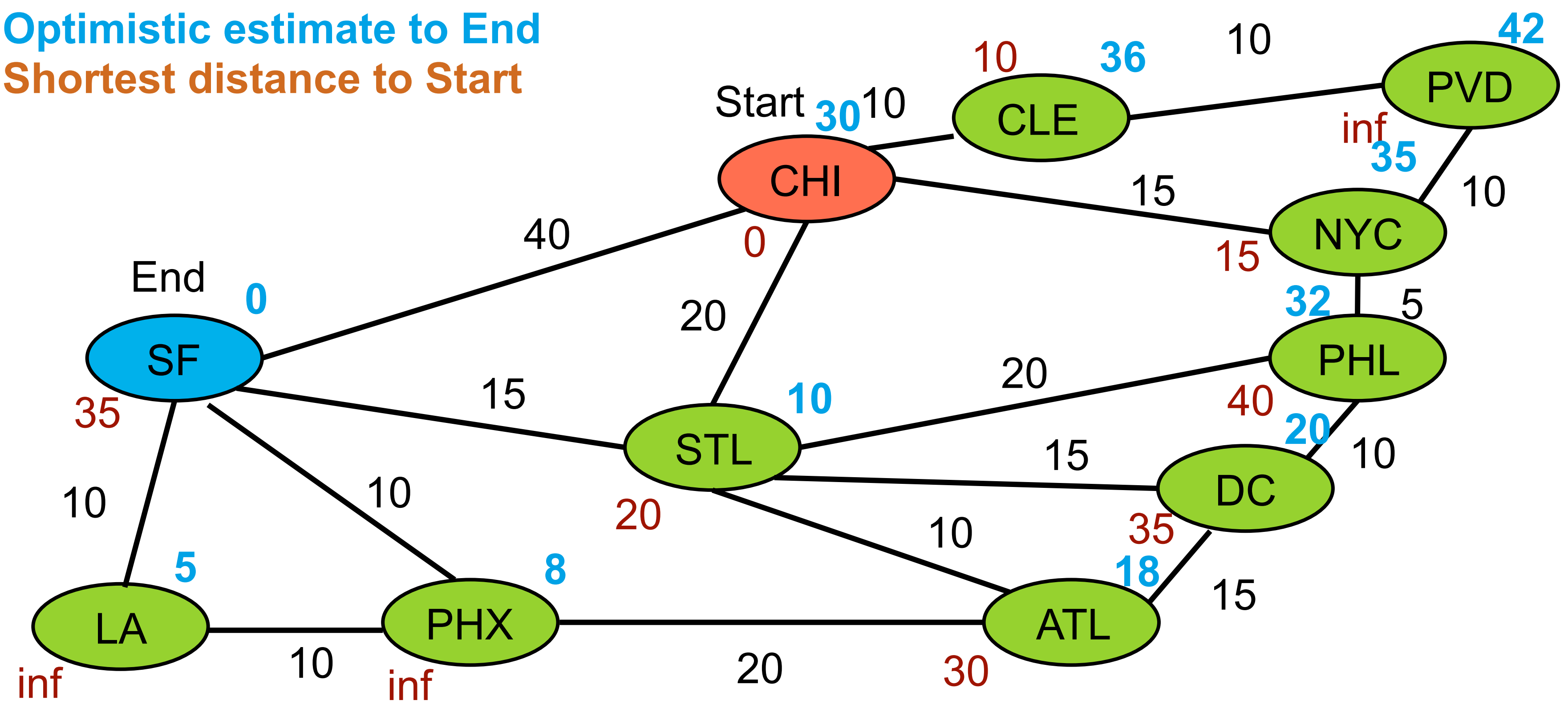
Optimistic Estimates



PQ: (STL,30) (SF,40) (CLE,46) (NYC,50) (…,inf)

Optimistic Estimates

Optimistic estimate to End
Shortest distance to Start



PQ: (SF,35) (CLE,46) (ATL,48) (NYC,50) (DC,55) (PHL,72) (…,inf)

Optimistic?

- It’s so important for estimates to be optimistic, ie never say it’s farther than it is, so that the best path will always be taken first
- Important to be **admissible:** a heuristic that never overestimates
- Inadmissible heuristics don’t always return the best path

A* (A Star) Algorithm

- The algorithm is as follows:
  - Decorate all nodes with:
    - Distance ∞, except start node, which has distance 0,
    - **Optimistic estimate of node-to-end-node distance.**
  - Add all nodes to a priority queue, with the **sum of both decorations as the priority**.
  - While the priority queue isn't empty **and you haven't expanded the destination**:
    - Remove the minimum priority node
    - Update the decorations and priority to the removed node's neighbors if the distances have decreased

A* Algorithm Pseuodocode

```
function astar(G, s, d):
  //Input: Graph G with vertices V, start vertex s, and destination vertex d
  //Output: u.dist set to shortest distance from s to u and path saved

   for v in V:   // Initialize vertices' decorations
        v.dist = infinity  // dist-from-start
        v.est  = heuristic(v, d) // dist-from-destination estimates
        v.prev = null // set previous pointers to null
   s.dist= 0

   PQ = PriorityQueue(V) // add all nodes with (v.dist + v.est) as priority
   while PQ not empty and PQ.getMin()!=d://haven't reached destination
      curr = PQ.removeMin()
      for all edges (curr, v):   //cost() returns edge weight
         if (curr.dist + cost(curr, v)) < v.dist: // if this path is shorter
            v.dist = curr.dist+cost(curr,v)  //Update to curr's path+weight
            v.prev = curr                   // Maintain pointers for path
            PQ.replaceKey(v, v.dist+v.est)
```

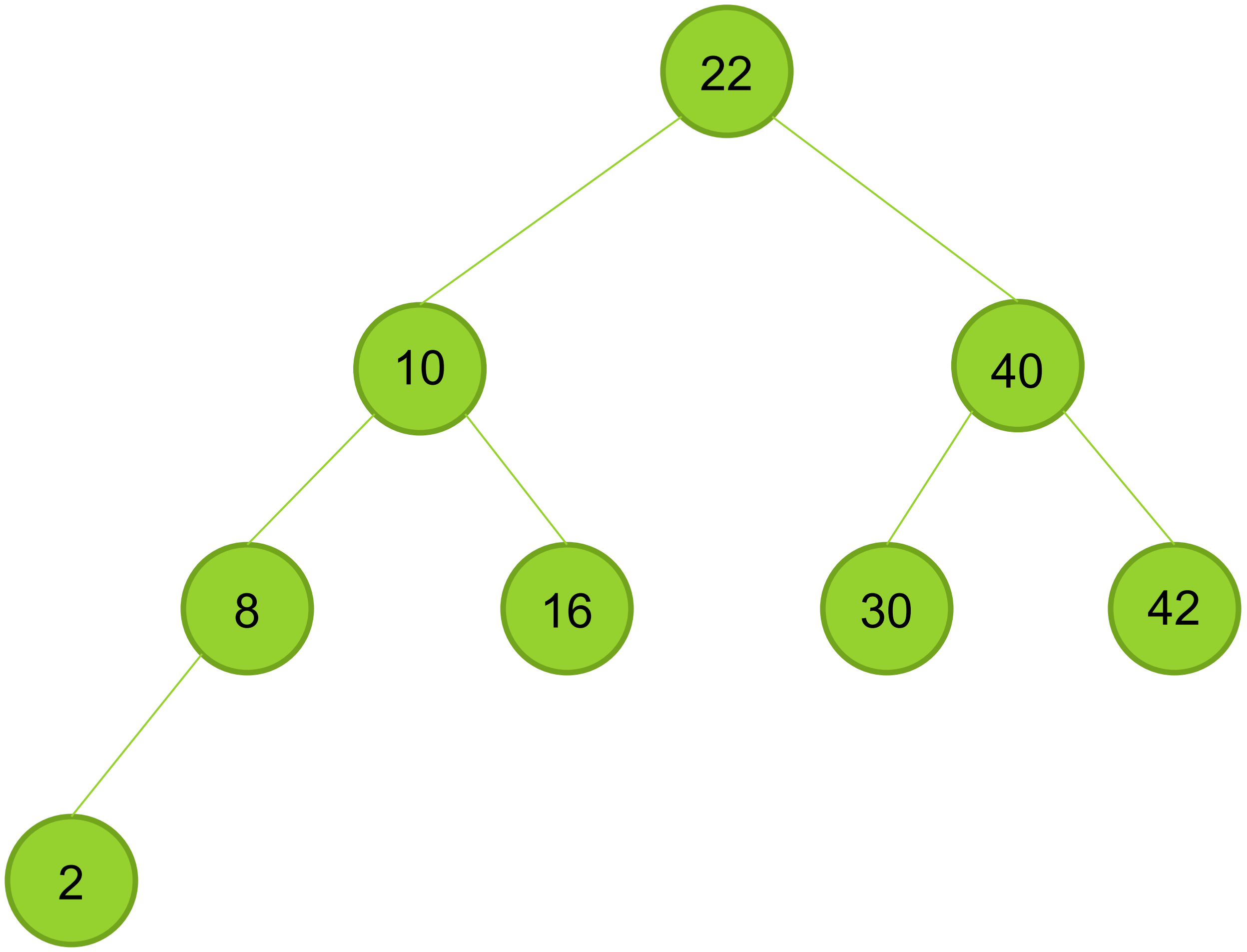
Other Applications and Heuristics

- What other problems do we use search to solve?
- Games like Chess, Go, Tic Tac Toe
- Dijkstras functions as BFS since the cost of all transitions are equal
- Heuristics can be much funkier than distance: chess could use number of your pieces on the board, sum of the values of pieces, moves until checkmate
- Runtime is dependent on difficulty of calculating heuristic

1D-Tree Example – Nearest Neighbor

2 8 10 16 22 30 40 42

1D-Tree Example – Nearest Neighbor



2D-Tree Example

- Now lets sort something a little more complicated, points in the Cartesian plane.
- A(10, 12), B(11, 9), C(3,4), D(5,3), E(7,8), F(1, 5)

2D-Tree Example

- Sort list in the 1st dimension(x)
  - F, C, D, E, A, B
- Center element becomes new node.

- Make two new lists, everything in the left list is less than or equal to the center, everything in the right list is greater than the center.

Left list: F, C, D

Right list: A, B

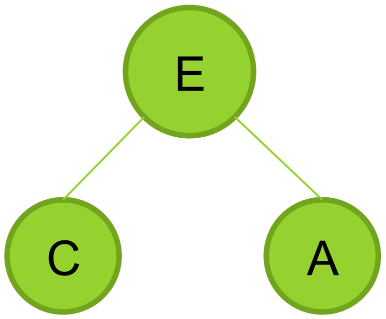
2D-Tree Example

- We now sort each list in the next dimension(y).

  F, C, D -> D, C, F

  A, B -> B, A

- Again we take the center element to be the new node



- Make our separate lists again.

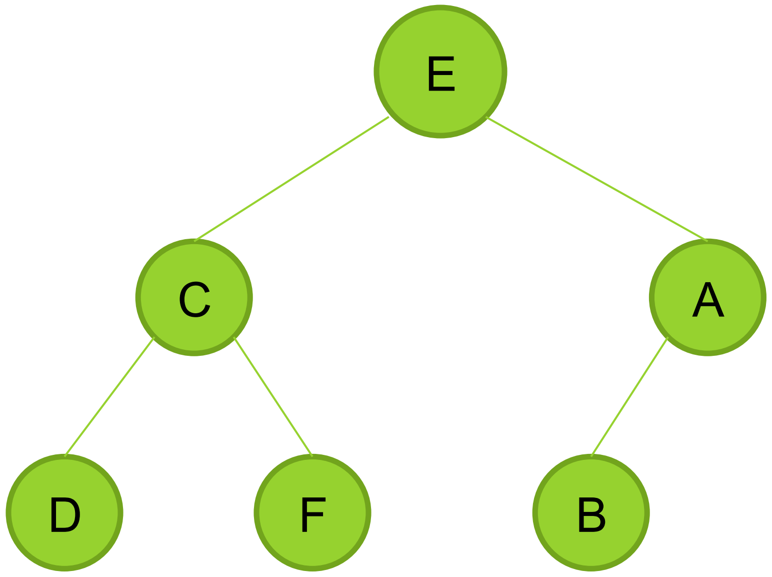
  D F B

2D-Tree Example

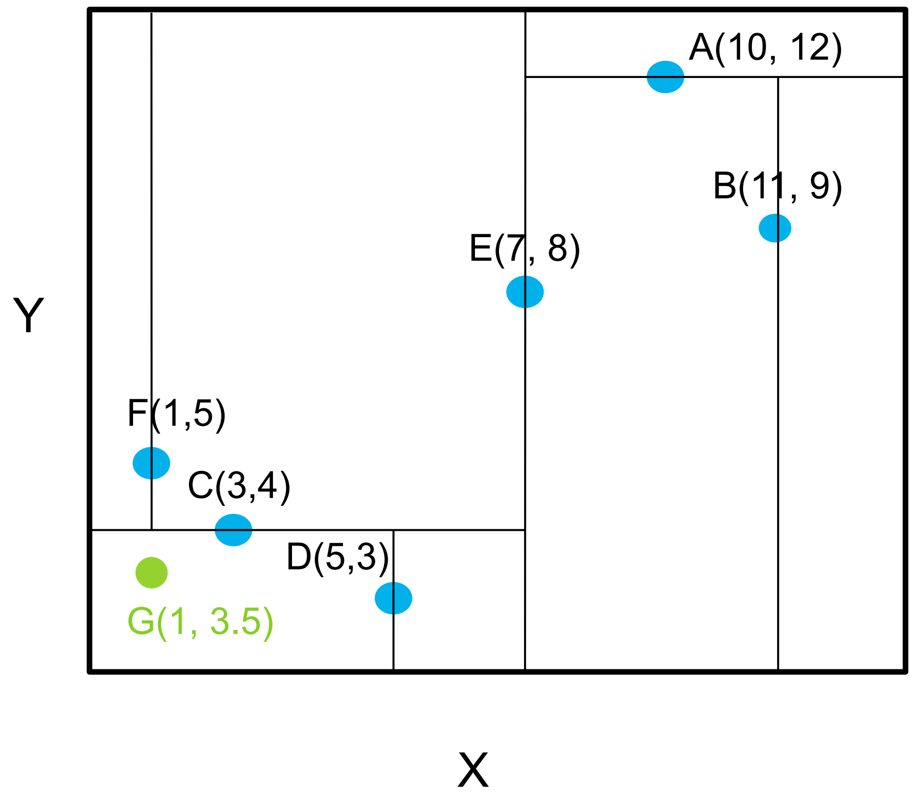
- Sort our lists; in the first dimension this time

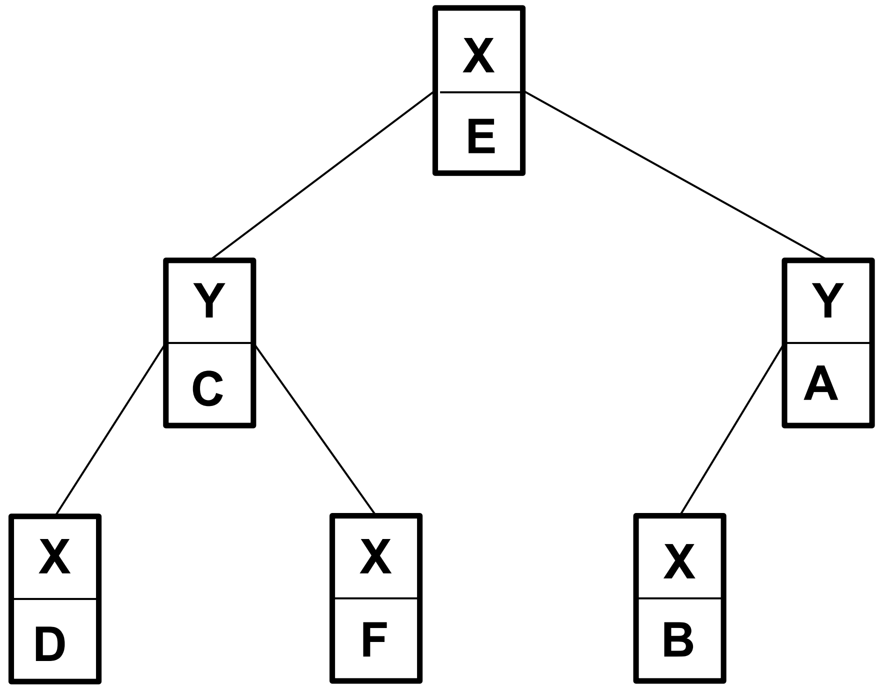
  D F B

- Make our new nodes

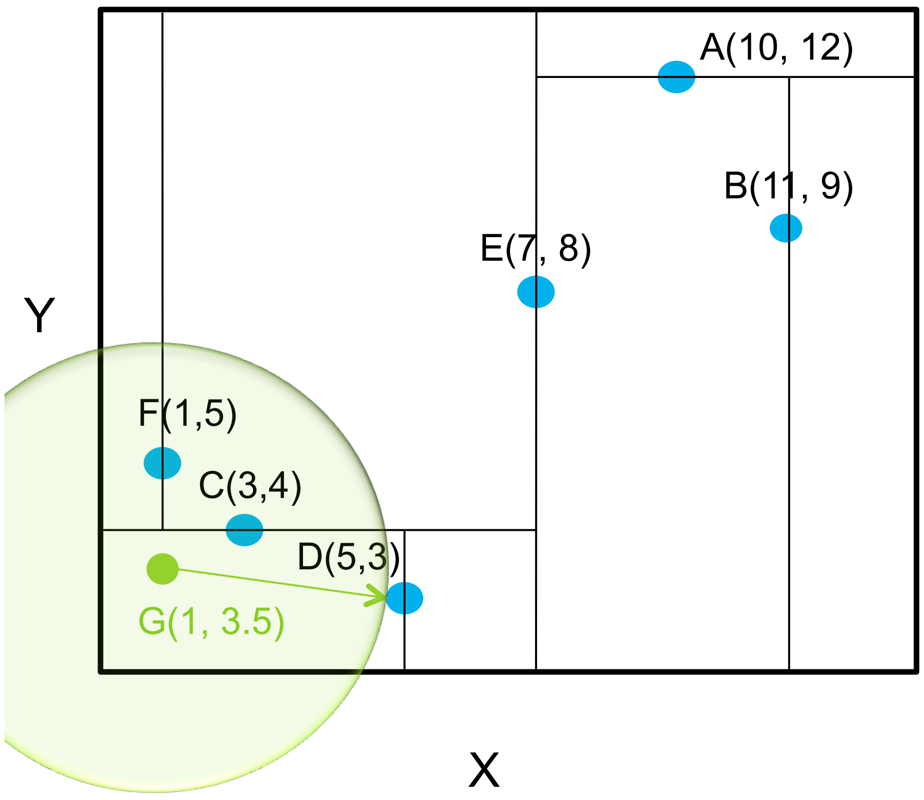


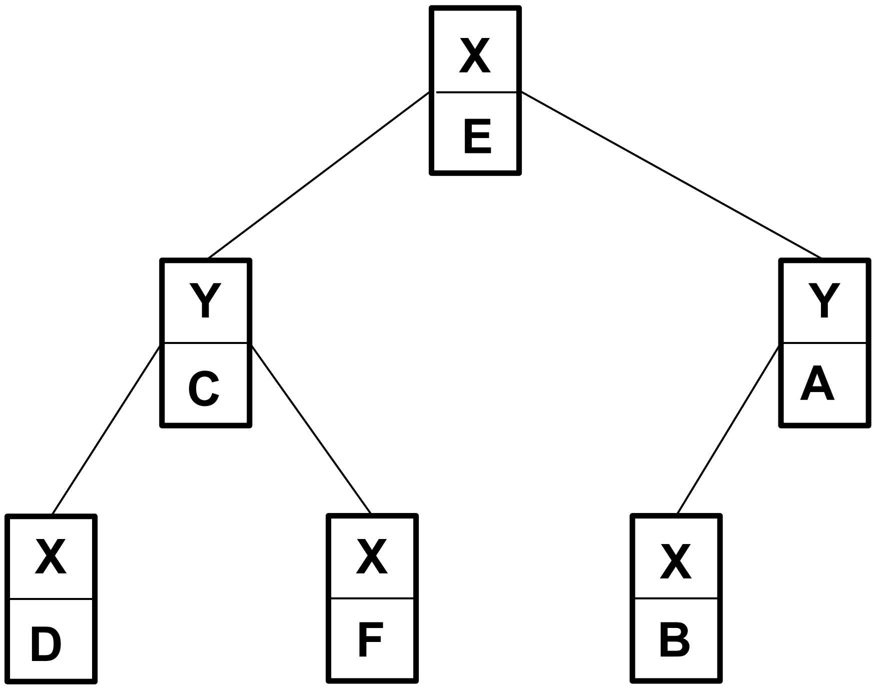
2D-Tree Example: Nearest Neighbor



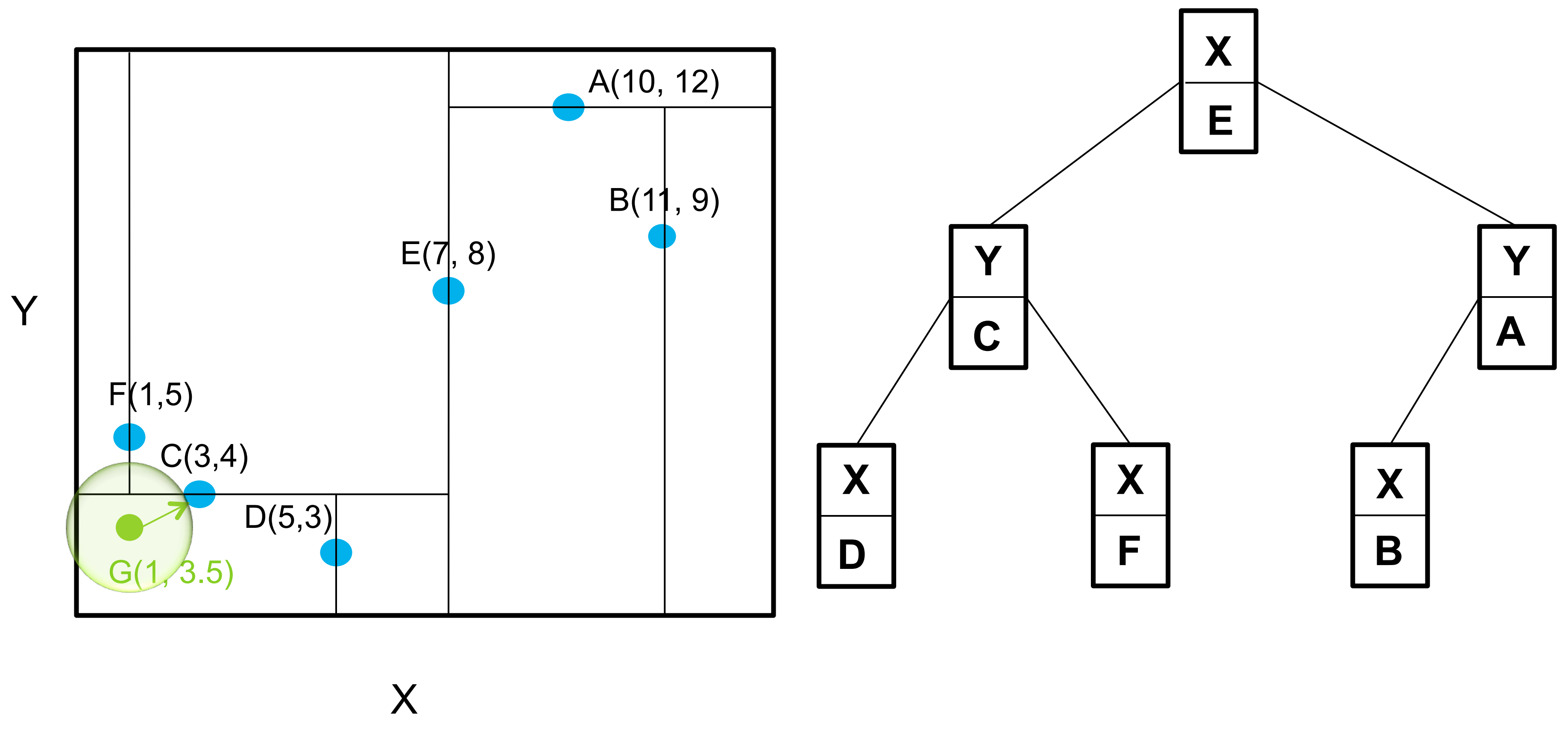


2D-Tree Example: Nearest Neighbor





2D-Tree Example: Nearest Neighbor

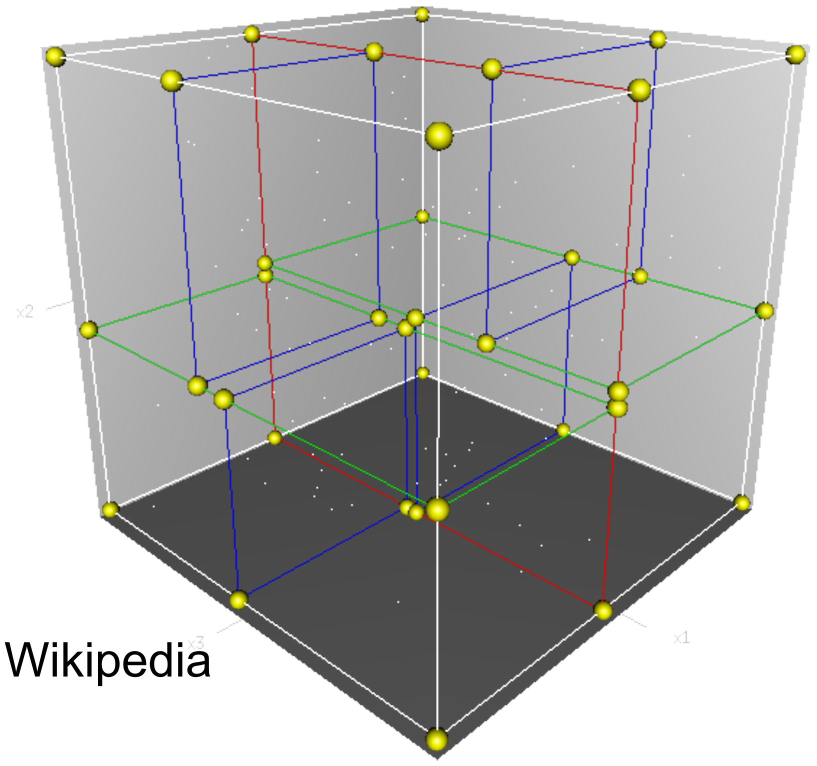


NNS: Pseudocode

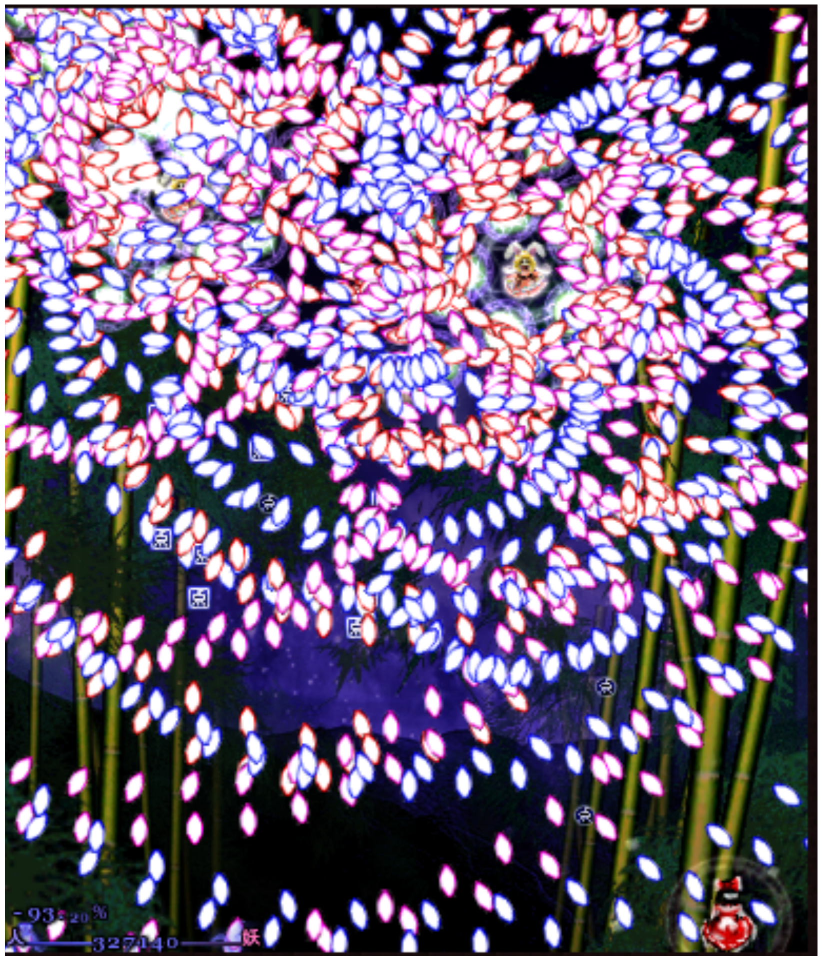
- Move down the tree recursively starting with the root. Go left or right depending on whether the point is less than or greater than the current node in the split dimension.
- Once you reach a leaf node, save it as the "current best"
- Unwind the recursion of the tree, performing the following steps at each node:
    - If the current node is closer than the current best, then it becomes the current best.
    - Check whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best.
        - This is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a comparison to see whether the difference between the splitting coordinate of the search point and current node is less than the distance from the search point to the current best.
        - If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so we must move down the other branch of the tree from the current node looking for closer points, following the same recursive process.
        - If the hypersphere doesn't intersect the splitting plane, then we continue walking up the tree, and the entire branch on the other side of that node is eliminated.

Wikipedia

3D-Tree example and Applications



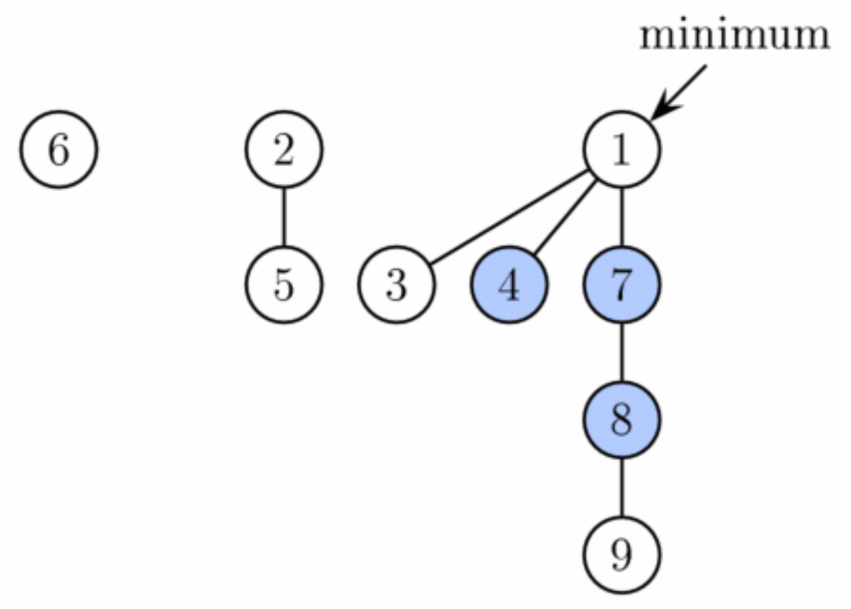
Wikipedia



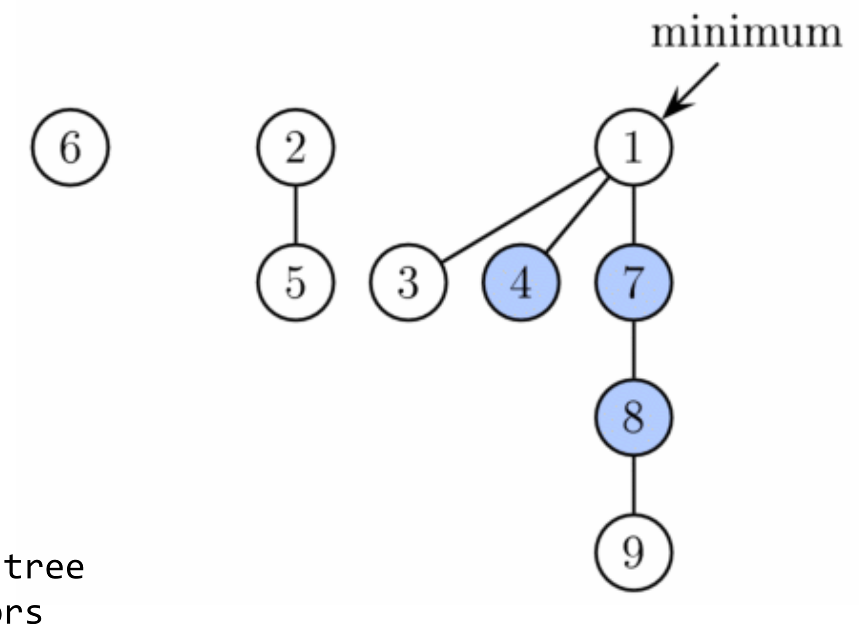
Fibonacci Heap

Operation	Binary Heap	Fibonacci Heap
insert	log(n)	1 (amortized)
decreaseKey	log(n)	1 (amortized)
removeMin	log(n)	log(n)

Fibonacci Heap DS: list of trees



Fibonacci Heap DS: list of trees

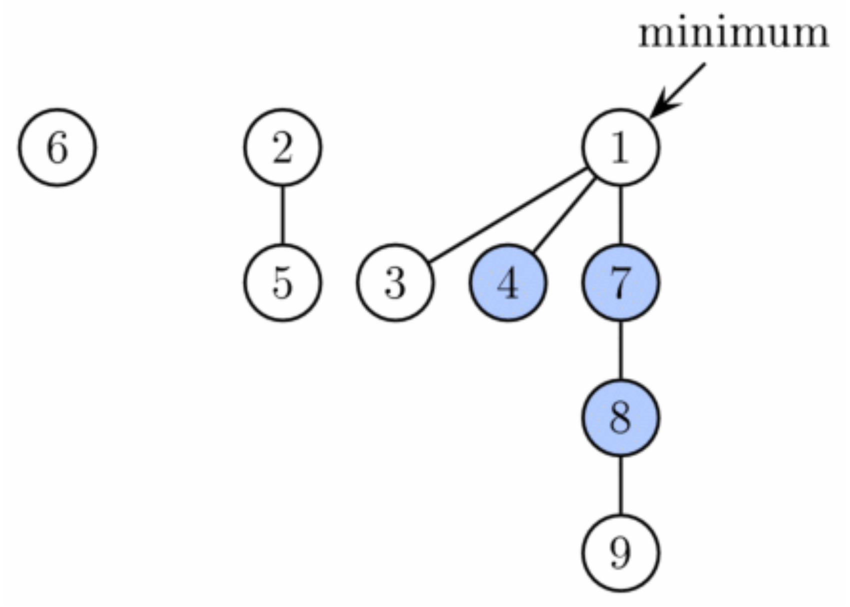


```
insert(k, p):
    Add a new tree with key and priority
    If (p < current min):
        point current min at new tree


decreasePri(n, p):
    decrease n.p
    if (n.p < n.parent.p):
        cut n from parent and make a new tree
        go up tree cutting marked ancestors
            until root or unmarked ancestor is reached
        mark first unmarked ancestor
        // note that roots are never marked
```

Fibonacci Heap DS: list of trees

```
removeMin():
    use pointer to get minimum
    make each child a new root
    combineTrees()
    point current min at new tree
```



Outline

- A* Shortest-path Search
- KD-trees for Nearest Neighbor
- Fibonacci Heap