Automatically enhancing locality in irregular applications

Milind Kulkarni
School of Electrical and Computer Engineering
PURDUE UNIVERSITY

Locality in irregular applications

- We have a good understanding of locality in *regular* applications
  - Operate over dense matrices and arrays
  - Many transformations to improve locality
    - Loop interchange, tiling, etc.
- Far less understanding of *irregular* applications
  - Operate over pointer-based structures

What's the problem?

- Irregular applications are complex!
  - Layout is dynamic → hard to find spatial locality
  - Access patterns are highly unpredictable → hard to find temporal locality
- Are there even common sources of locality in irregular programs?

Gameplan

- Focus on subset of irregular applications to find common patterns
- Develop models for reasoning about locality
- Design transformations to improve locality
- Determine correctness criteria
- Implement automatic, tuned transformations
- Rinse and repeat

Gameplan

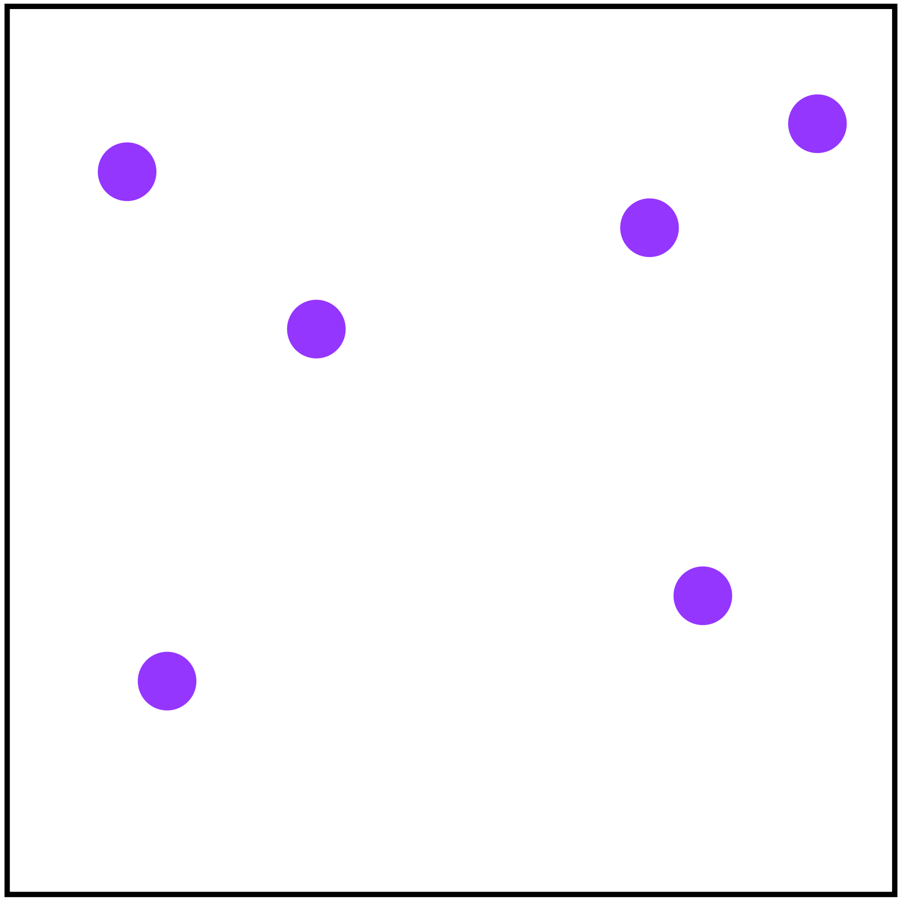
- Focus on subset of irregular applications to find common patterns
- … lity
- … y
- …
- … tuned transformations
- Rinse and repeat

- Focus on tree traversal algorithms
- 2 transformations: *point blocking* and *traversal splicing*.
- Automatic transformations and tuning
- Performance improvements of >200% (pb) and >400% (ts)

Narrow the scope

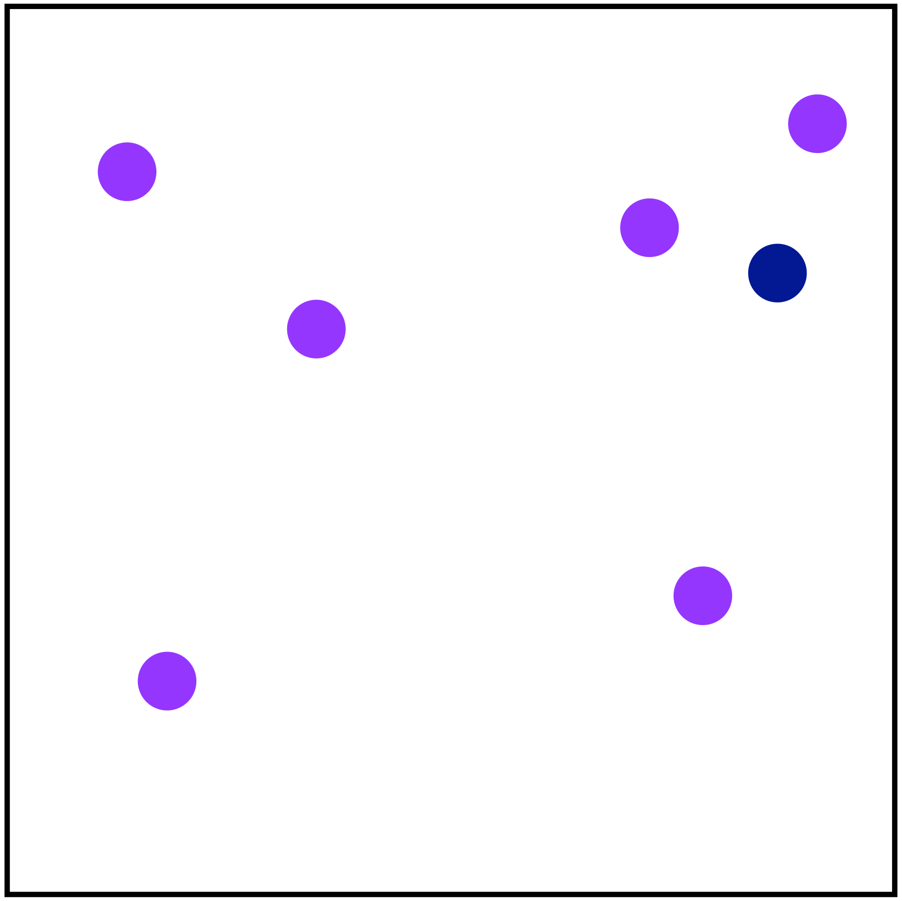
- We focus on a restricted set of irregular applications: *tree traversal algorithms*
- Appear in numerous domains:
  - Scientific: Barnes-Hut
  - Graphics: bounding-volume hierarchies, lightcuts
  - Data-mining: nearest-neighbor, point correlation
- Key feature: recursive traversals of tree structure
  - Repeated traversals → opportunity for locality!

Point correlation



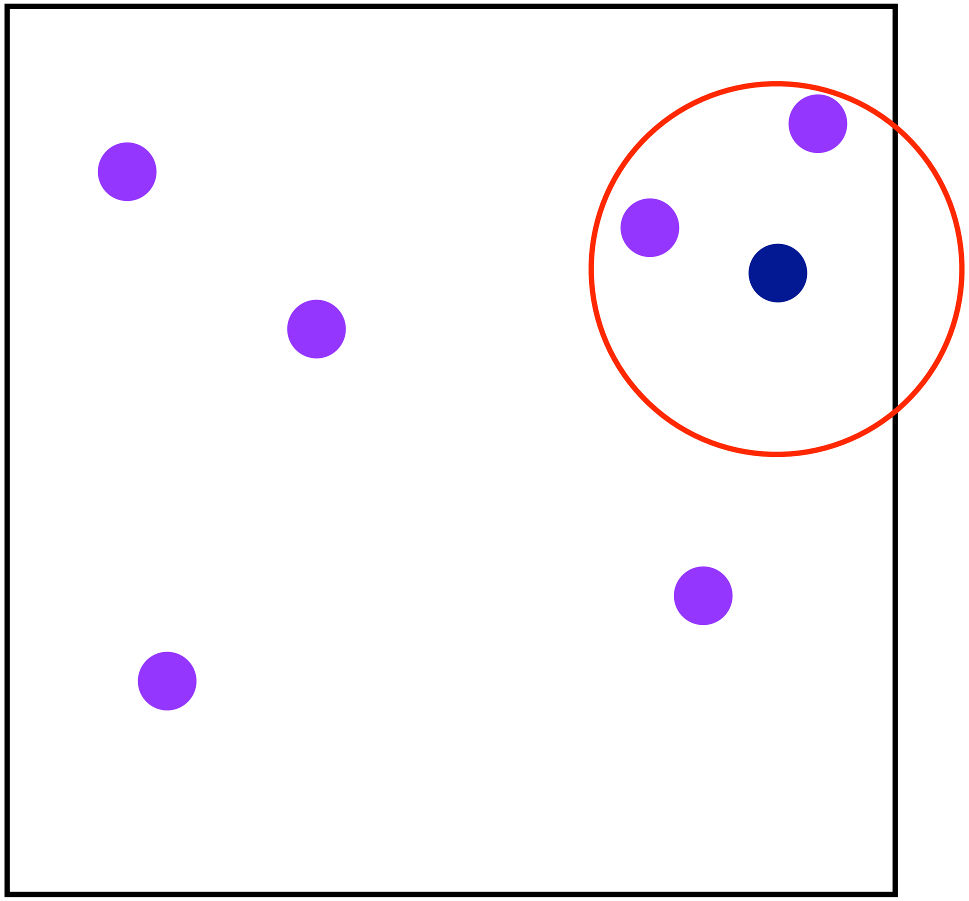
- Data mining algorithm
- Goal: given a set of ***N*** points in *k* dimensions and a point ***p***, find all points within a radius ***r*** of ***p***
- Naïve approach: compare all ***N*** points with ***p***
- Better approach: build *kd-tree* over points, traverse tree for point ***p***, prune subtrees that are far from ***p***

Point correlation



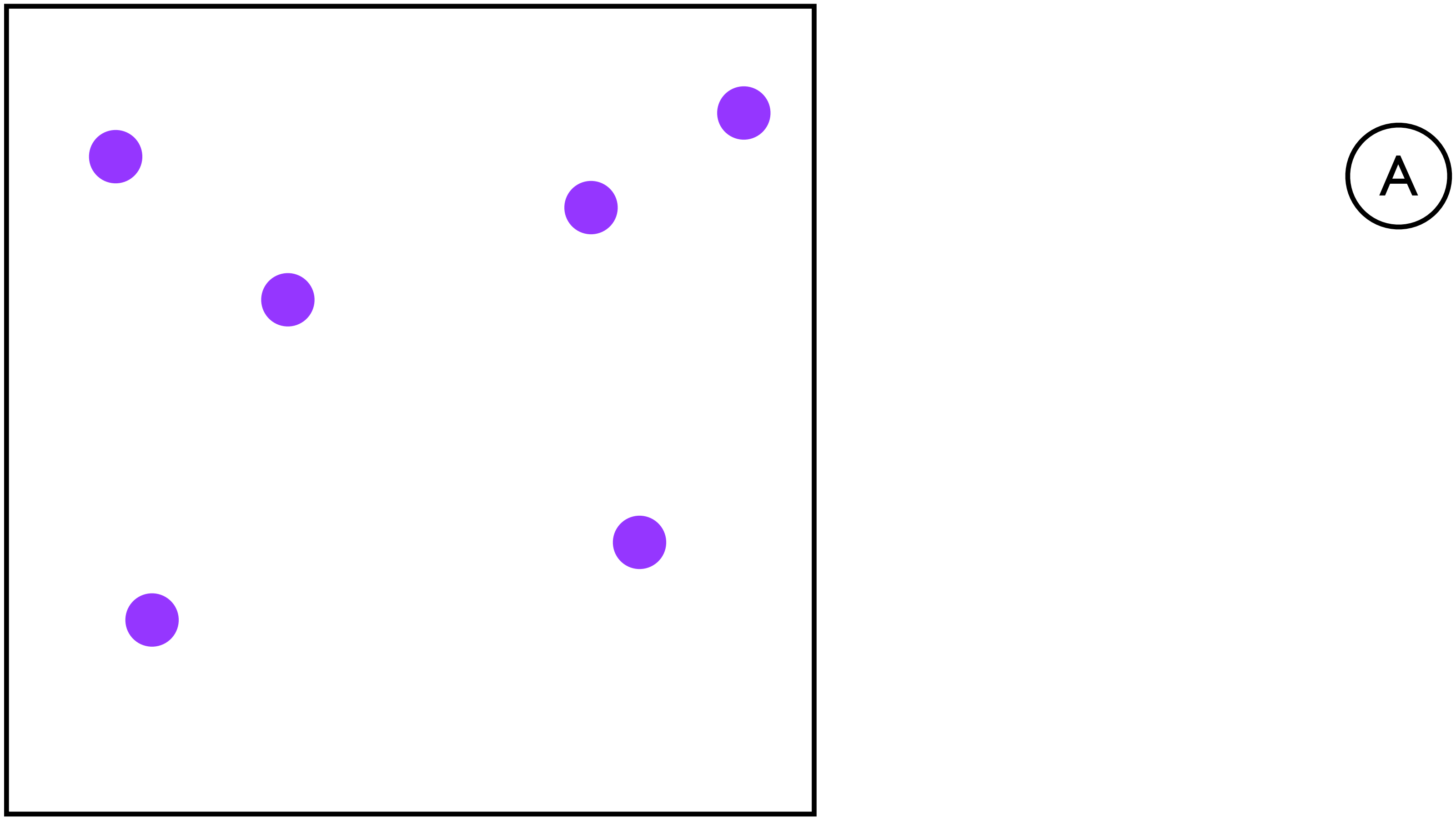
- Data mining algorithm
- Goal: given a set of ***N*** points in *k* dimensions and a point ***p***, find all points within a radius ***r*** of ***p***
- Naïve approach: compare all ***N*** points with ***p***
- Better approach: build *kd-tree* over points, traverse tree for point ***p***, prune subtrees that are far from ***p***

Point correlation

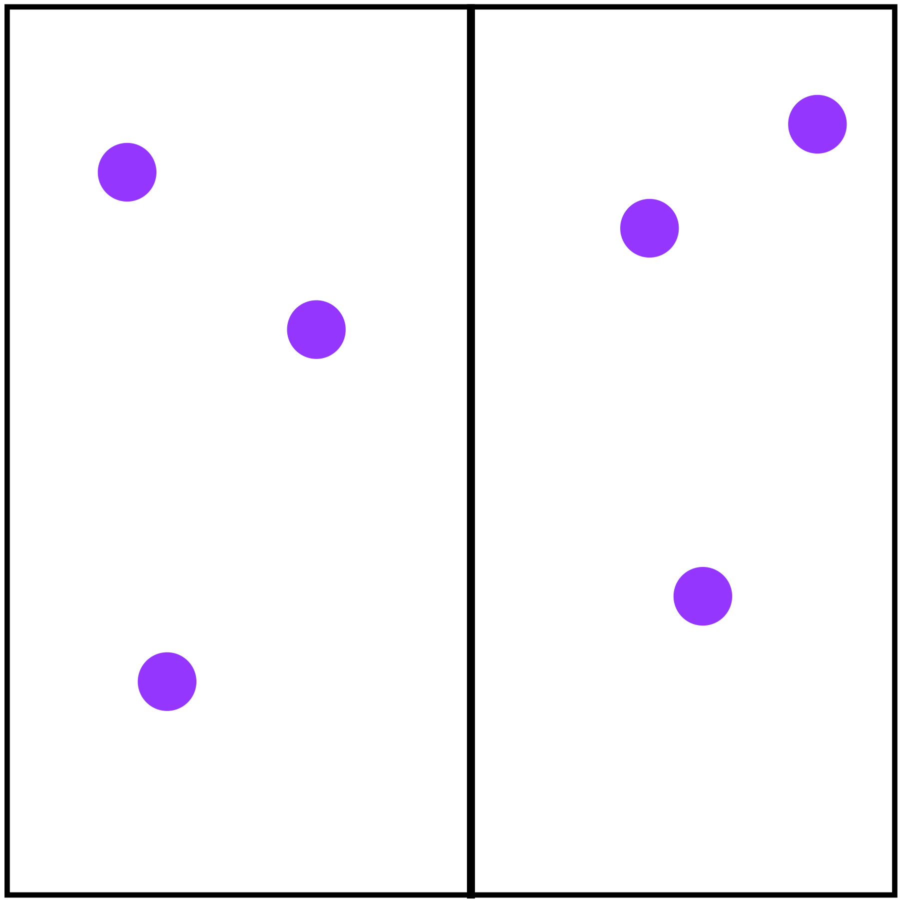


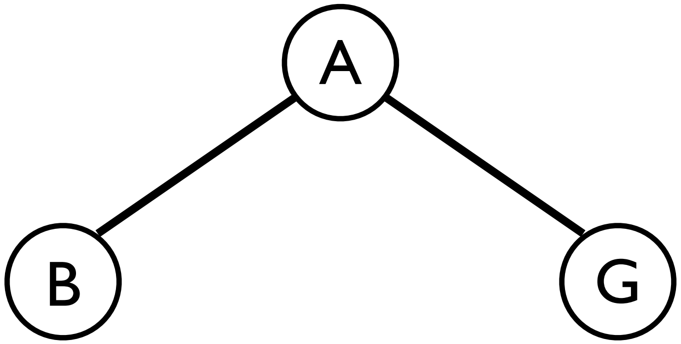
- Data mining algorithm
- Goal: given a set of ***N*** points in *k* dimensions and a point ***p***, find all points within a radius ***r*** of ***p***
- Naïve approach: compare all ***N*** points with ***p***
- Better approach: build *kd-tree* over points, traverse tree for point ***p***, prune subtrees that are far from ***p***

Point correlation

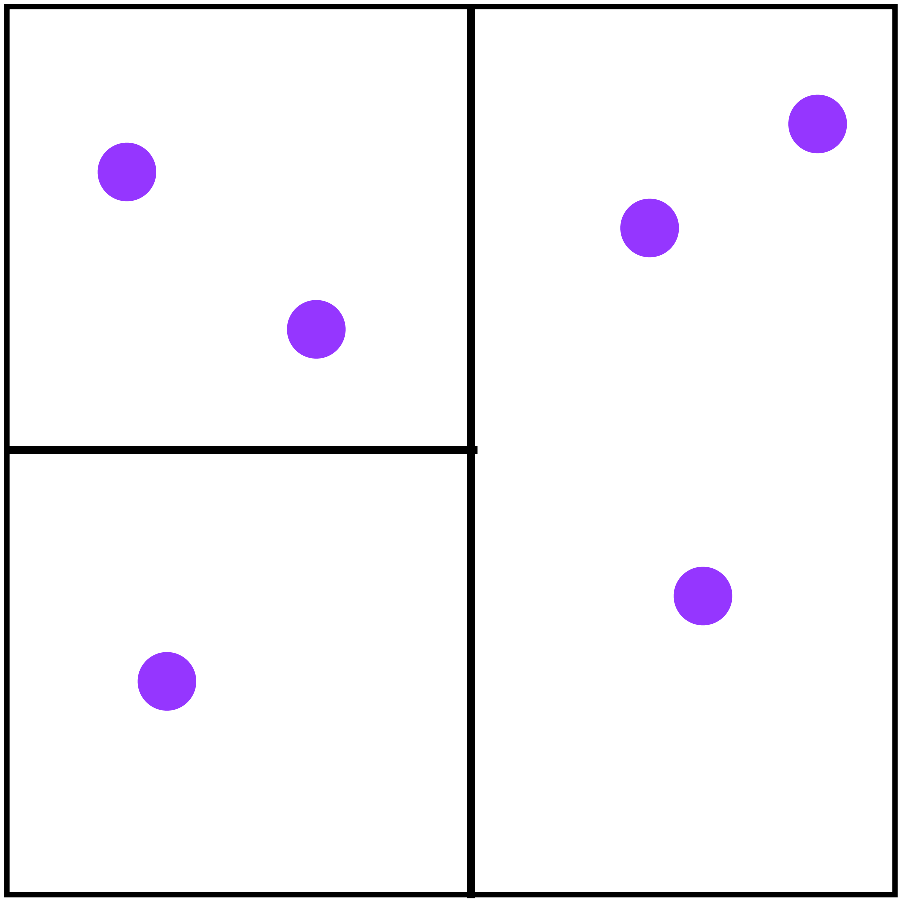


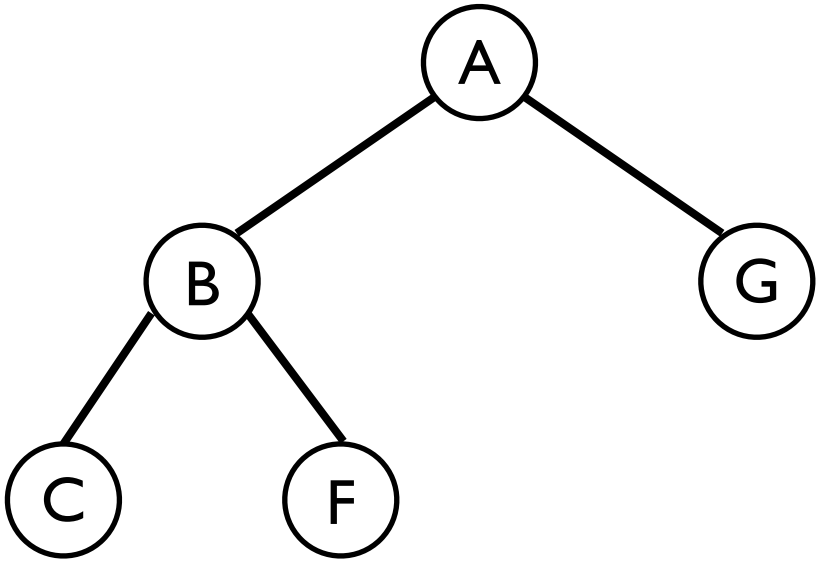
Point correlation



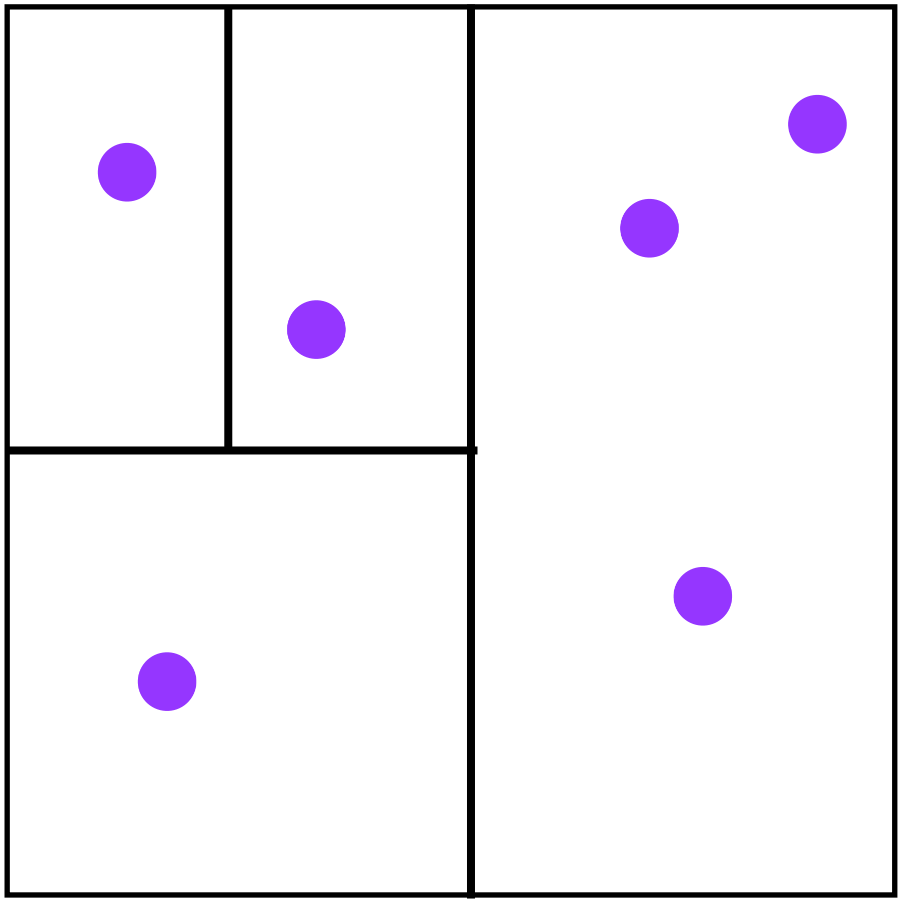


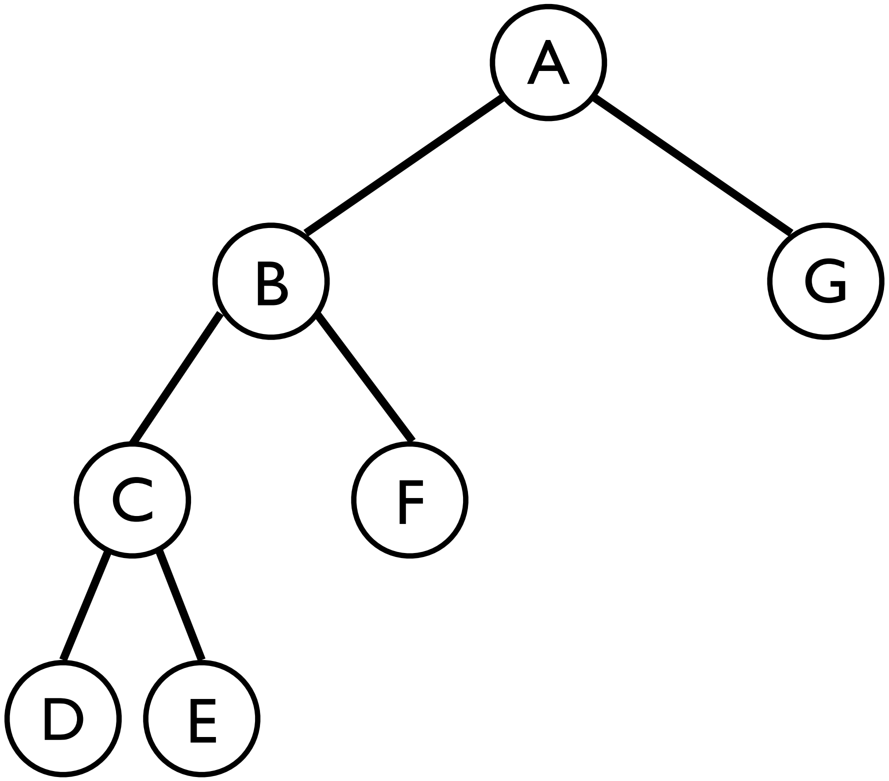
Point correlation



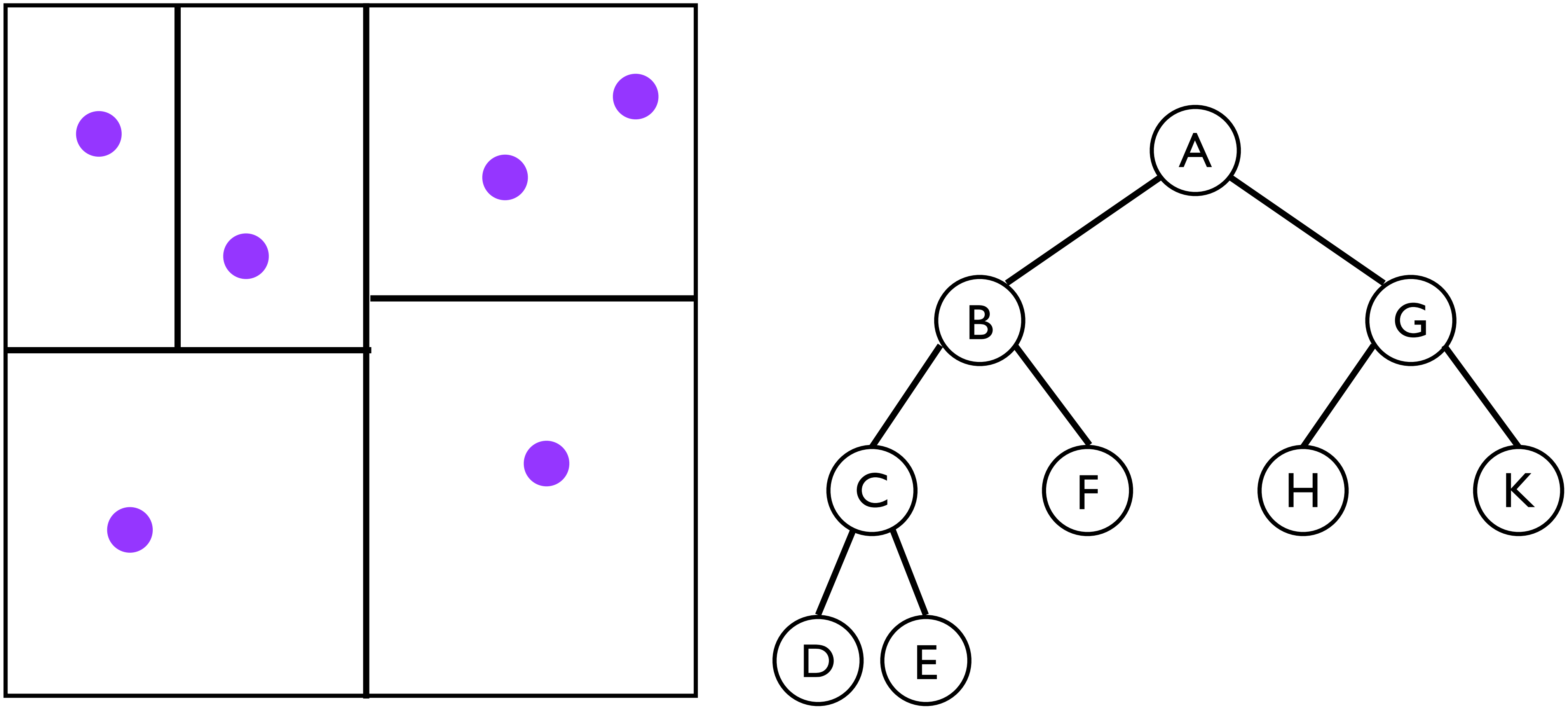


Point correlation

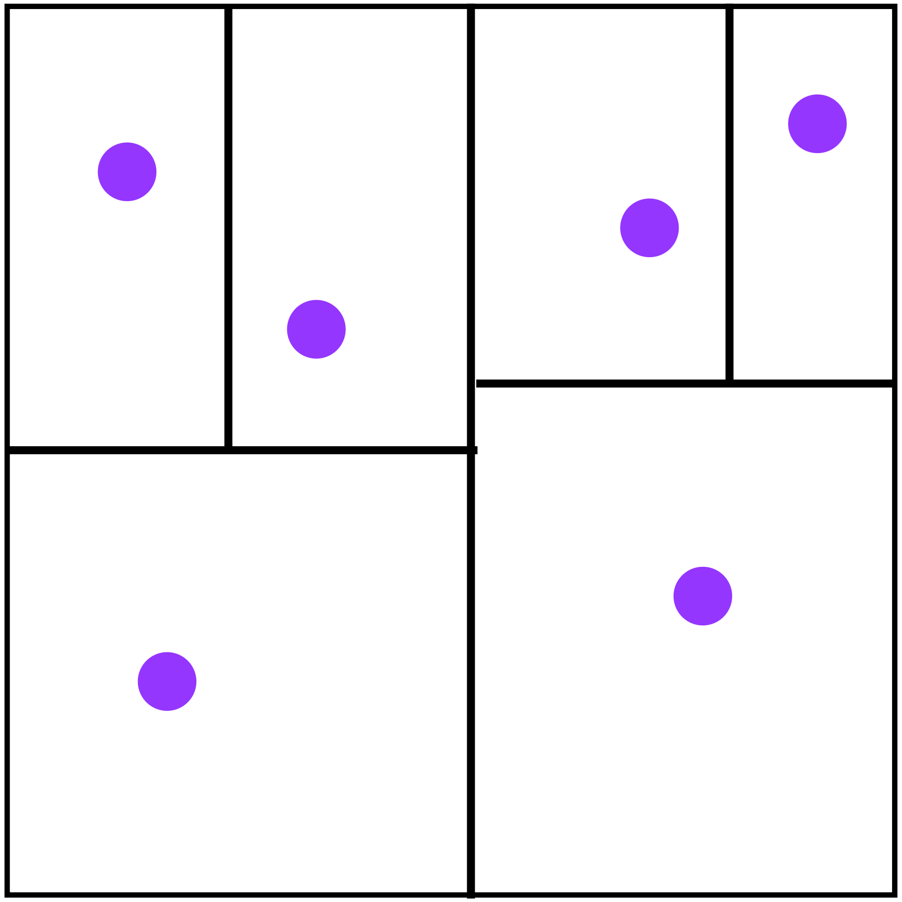


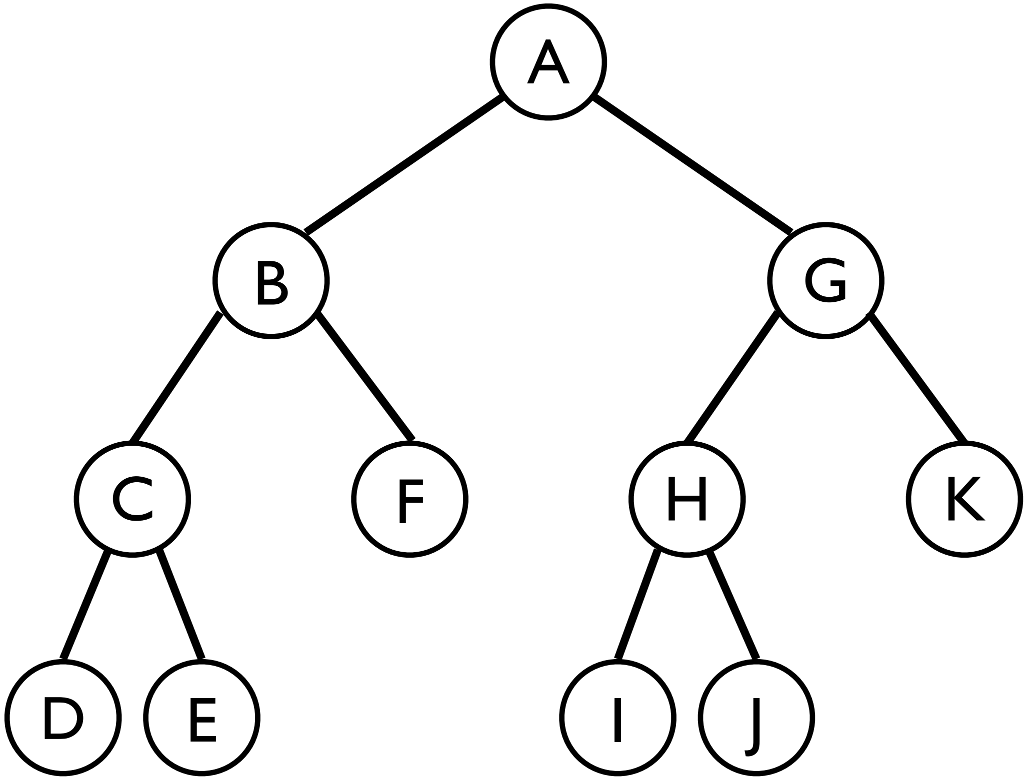


Point correlation

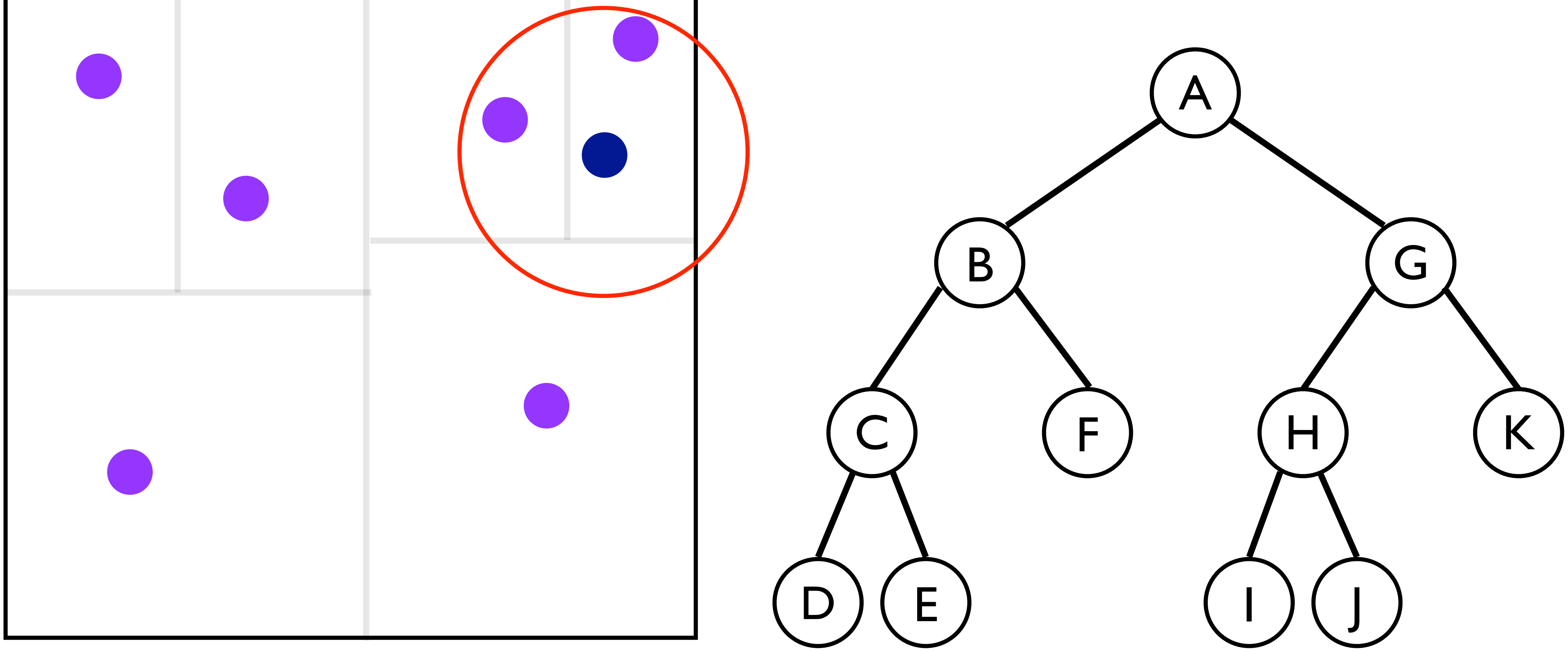


Point correlation

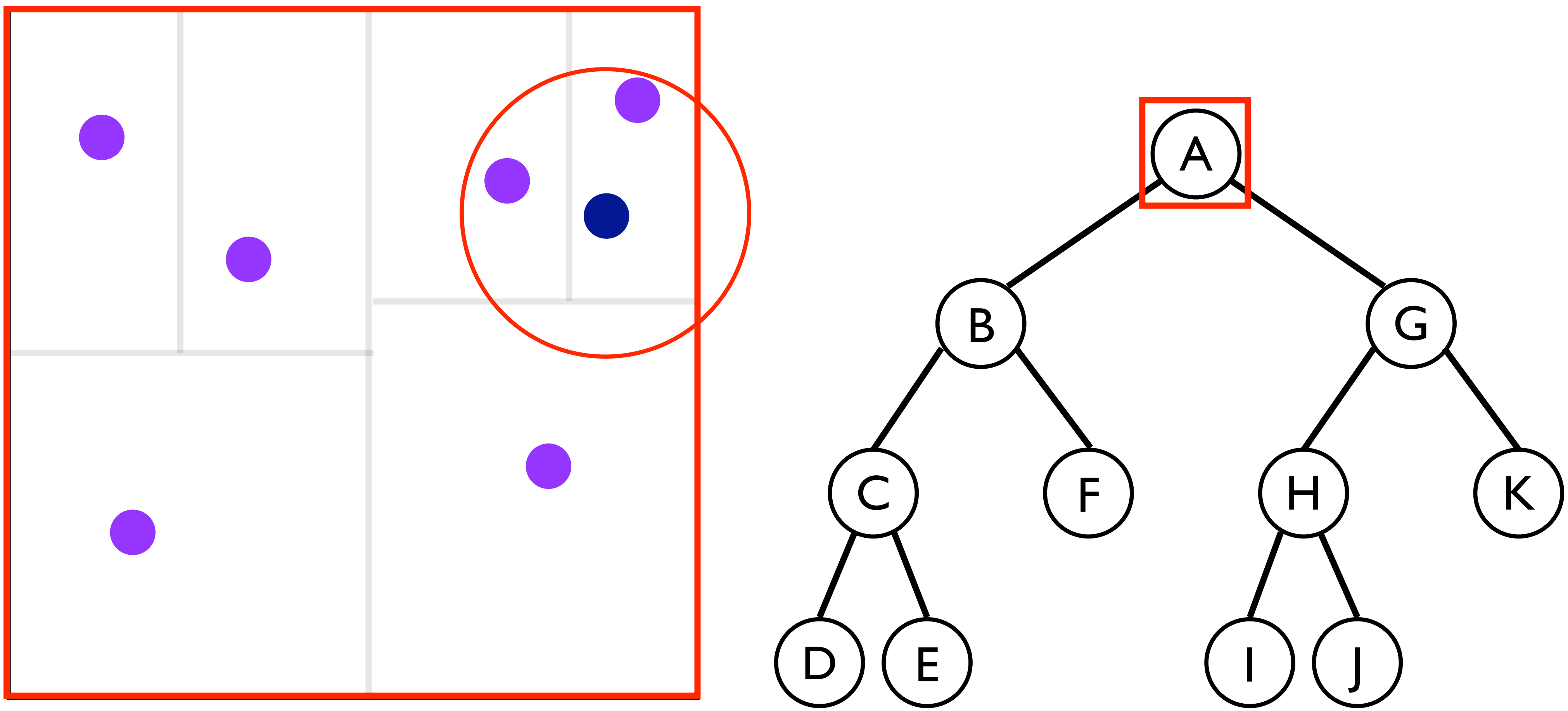




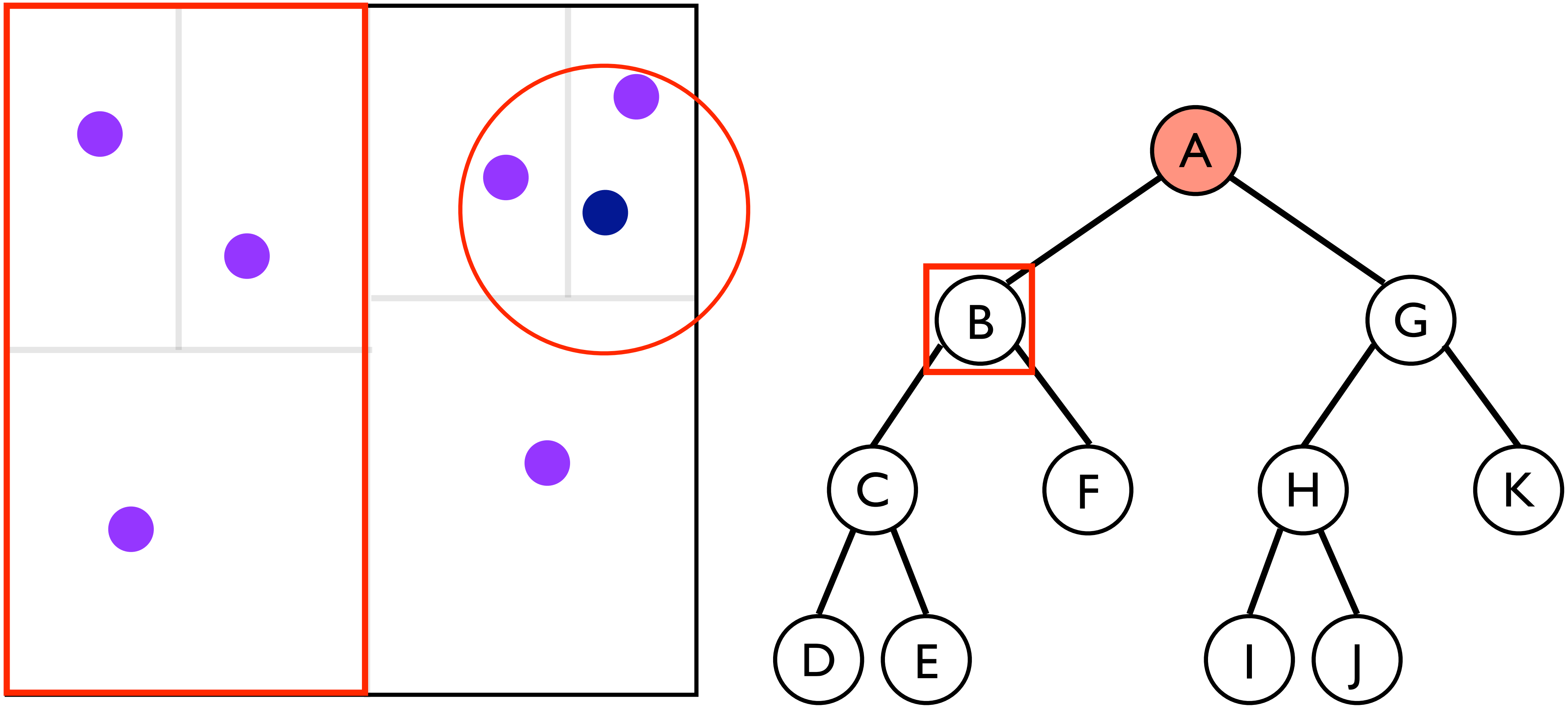
Point correlation



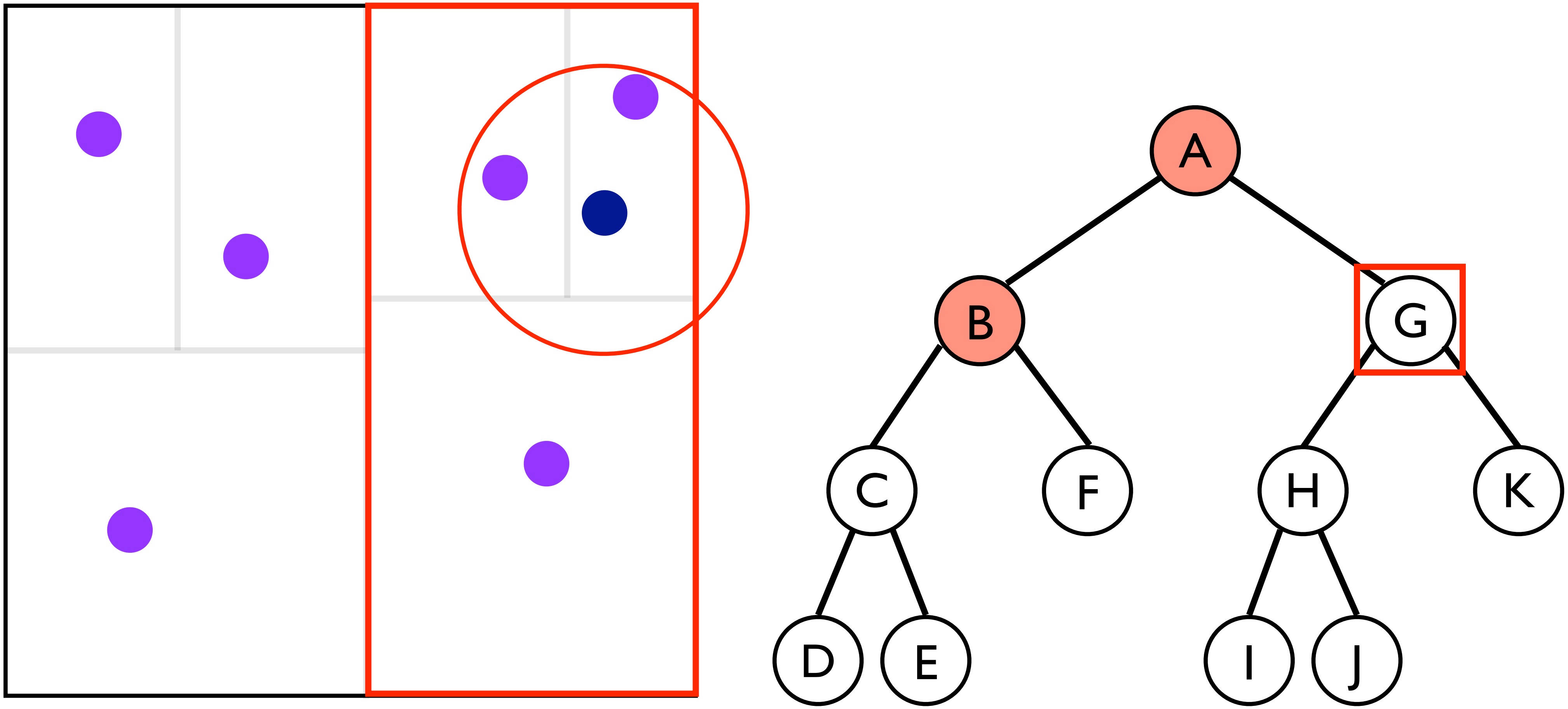
Point correlation



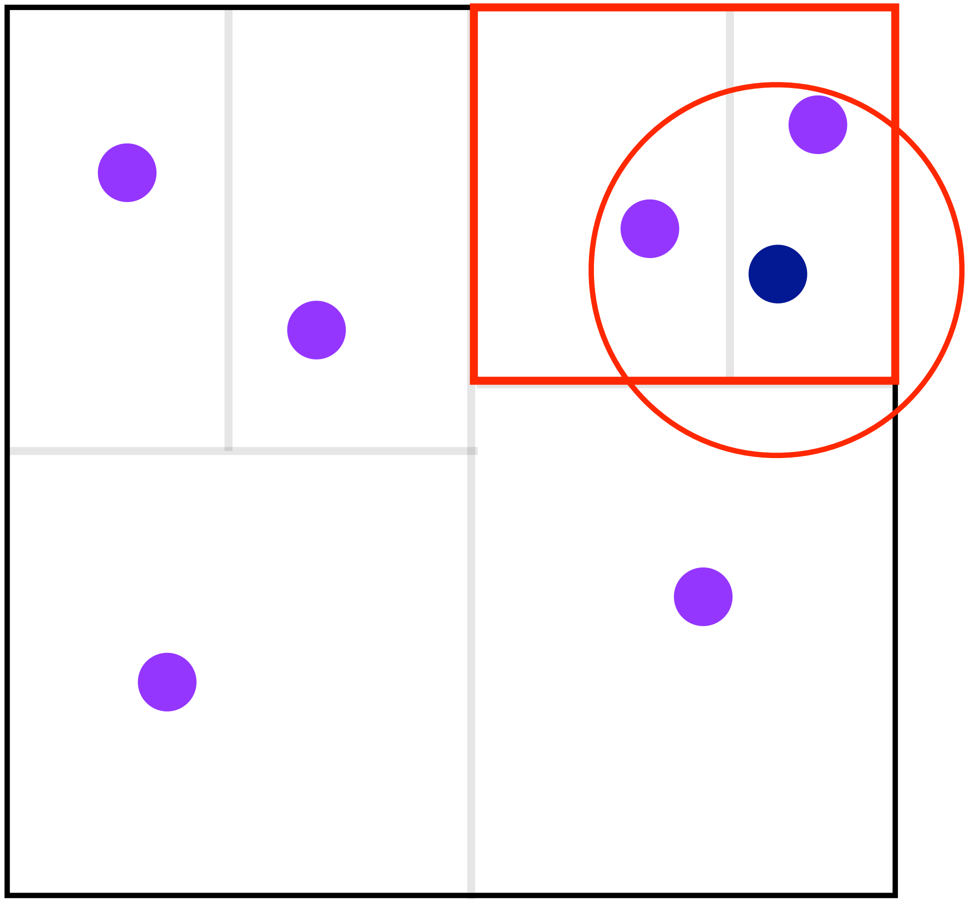
Point correlation

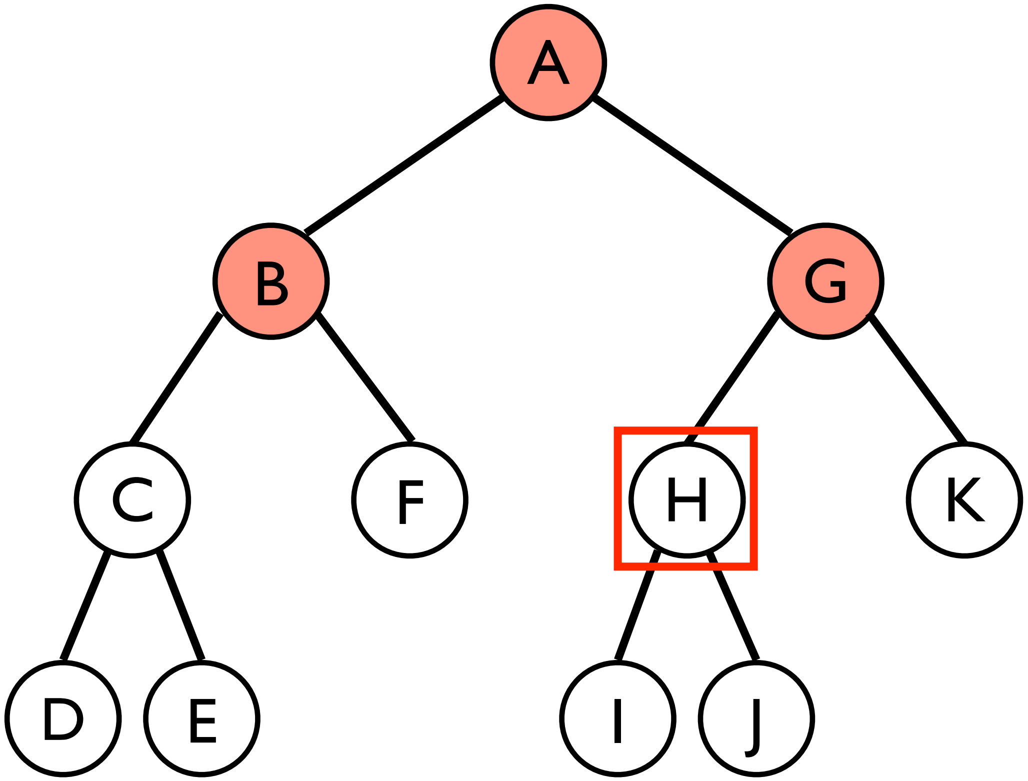


Point correlation

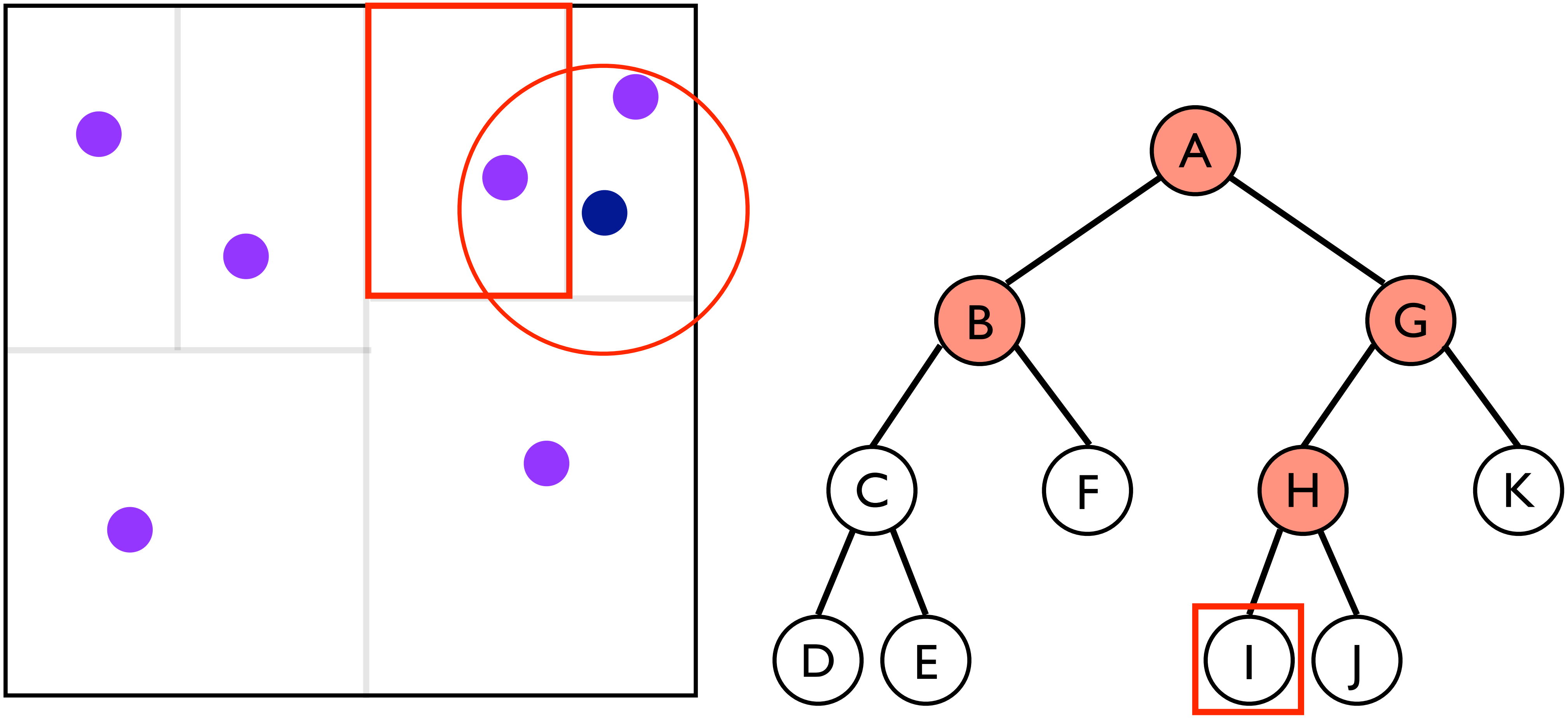


Point correlation

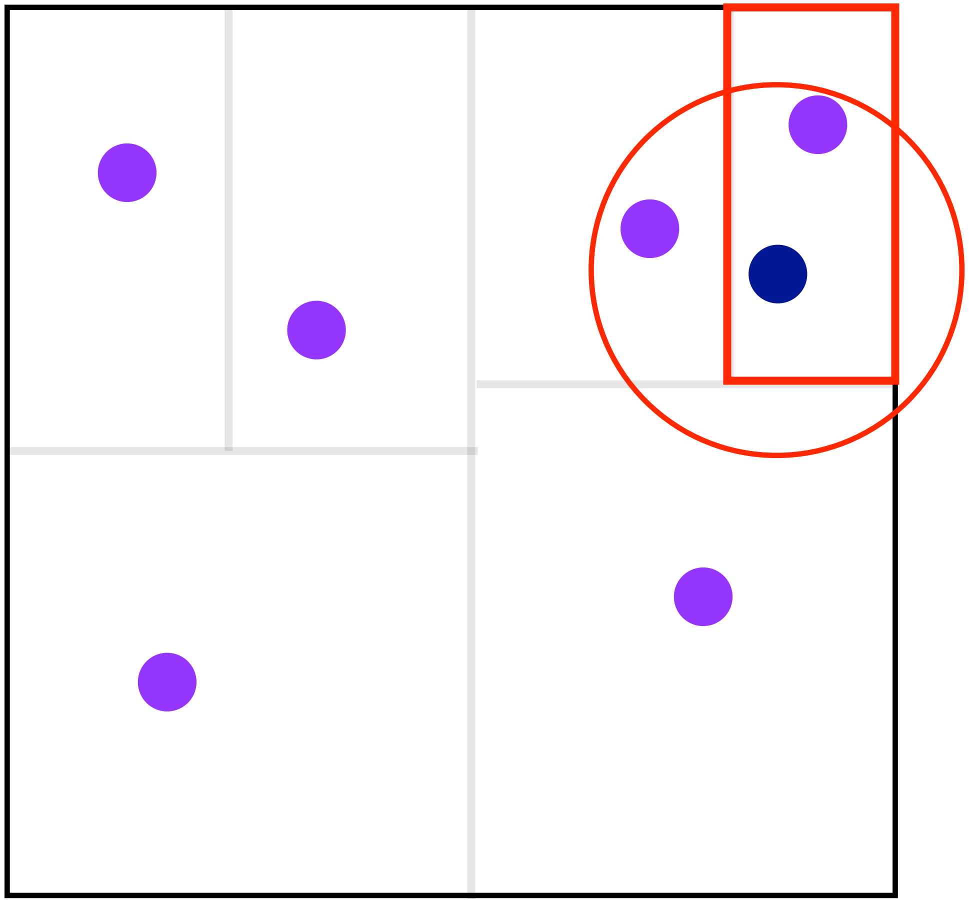


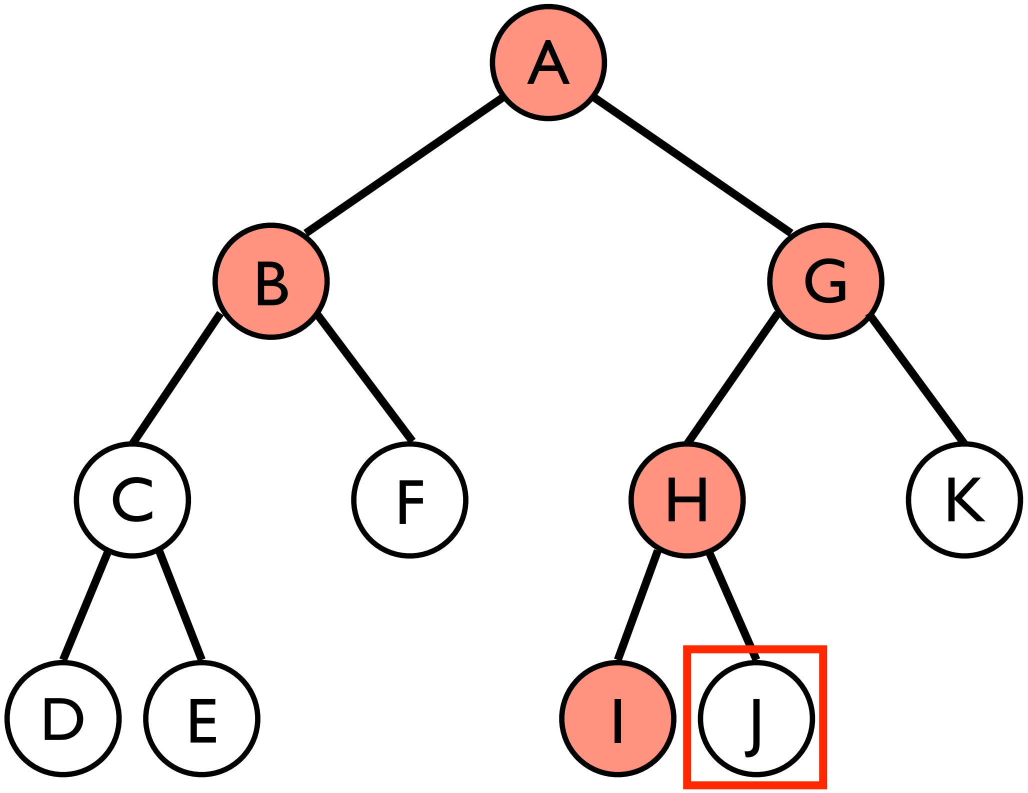


Point correlation

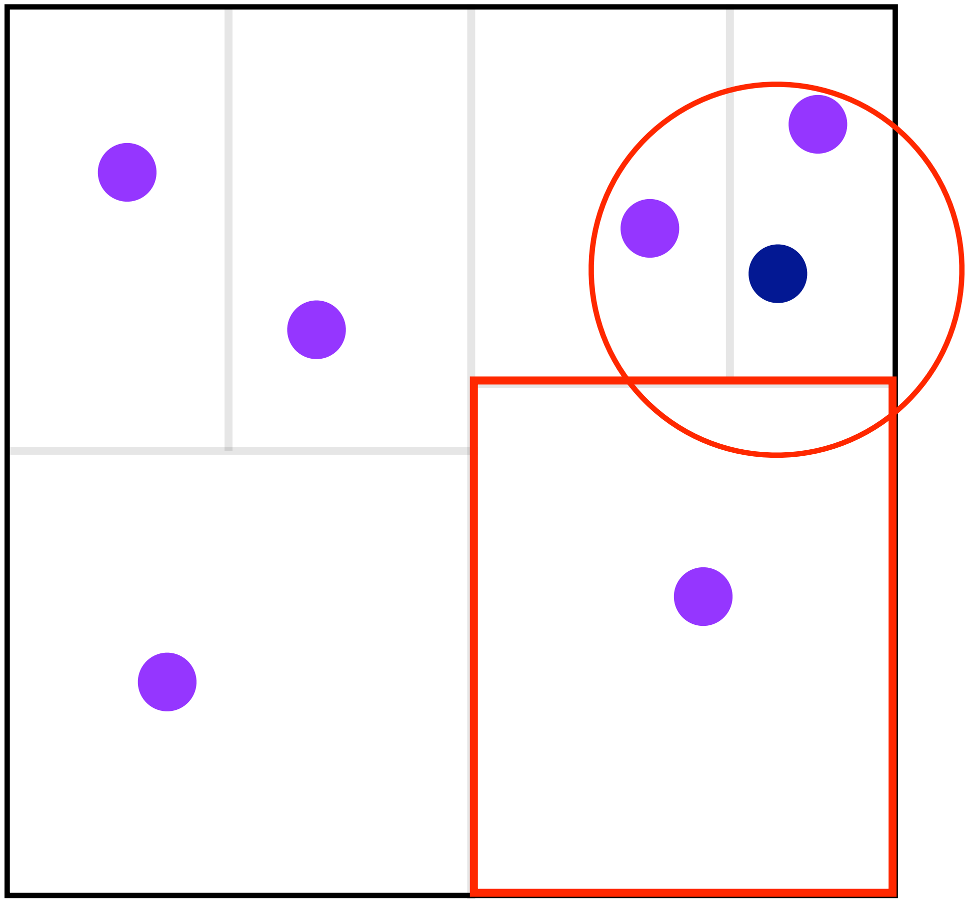


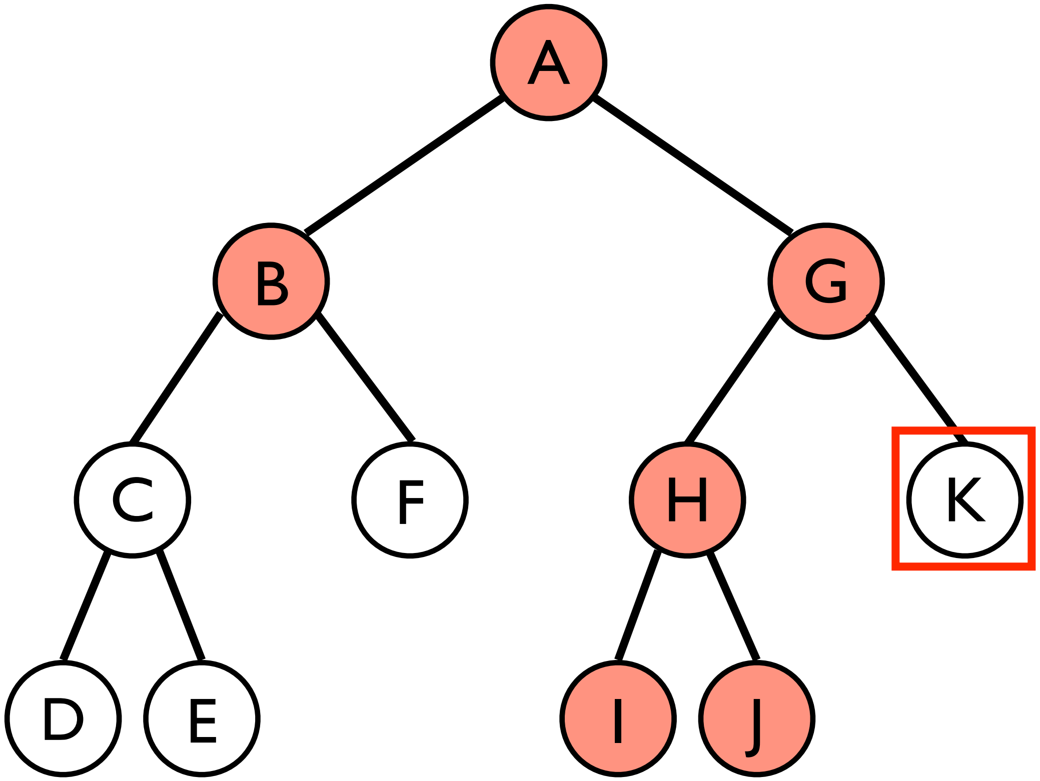
Point correlation





Point correlation





Point correlation

```
KDCell root = /* build kdtree */;
Set<Point> ps;
double radius;

foreach Point p in ps {
   recurse(p, root, radius);
}
...
void recurse(Point p, KDCell node, double r) {
   if (tooFar(p, node, r)) return;
   if (node.isLeaf() && (dist(node.point, p) < r))
      p.correlated++;
   else {
      recurse(p, node.left, r);
      recurse(p, node.right, r);
   }
}
```

Basic pattern

```
TreeNode root;
Set<Point> ps;

foreach Point p in ps {
   recurse(p, root, ...);
}
...
recurse(Point p, KDCell node, ...) {
   if (truncate?(p, node, ...))
      { ... }
   recurse(p, node.child1, ...);
   recurse(p, node.child2, ...);
   ...
}
```

Basic pattern

```
TreeNode root;
Set<Point> ps;

foreach Point p in ps {
   recurse(p, root, ...);
}
...
recurse(Point p, KDCell node, ...) {
   if (truncate?(p, node, ...))
      { ... }
   recurse(p, node.child1, ...);
   recurse(p, node.child2, ...);
   ...
}
```

recursive traversal

Basic pattern

```
TreeNode root;
Set<Point> ps;

foreach Point p in ps {
   recurse(p, root, ...);
}
...
recurse(Point p, KDCell node, ...) {
   if (truncate?(p, node, ...))
      { ... }
   recurse(p, node.child1, ...);
   recurse(p, node.child2, ...);
   ...
}
```

recursive structure

recursive traversal

Basic pattern

```
TreeNode root;
Set<Point> ps;

foreach Point p in ps {
   recurse(p, root, ...);
}
...
recurse(Point p, KDCell node, ...) {
   if (truncate?(p, node, ...))
      { ... }
   recurse(p, node.child1, ...);
   recurse(p, node.child2, ...);
   ...
}
```

recursive structure

repeated traversal

recursive traversal

Refined goal

- Improve temporal locality in *repeated recursive traversals of recursive structures*

Gameplan

- Focus on subset of irregular applications to find common patterns
- Develop models for reasoning about locality
- Design transformations to improve locality
- Determine correctness criteria
- Implement automatic, tuned transformations
- Rinse and repeat

An abstract model

- Irregular traversals are tricky due to all the pointer chasing and recursion.

```
foreach Point p in ps {
   recurse(p, root, ...);
}
...
recurse(Point p, KDCell node, ...) {
   if (truncate?(p, node, ...))
      { ... }
   recurse(p, node.child1, ...);
   recurse(p, node.child2, ...);
   ...
}
```

An abstract model

- Irregular traversals are tricky due to all the pointer chasing and recursion.

```
foreach Point p in ps {
   recurse(p, root, ...);
}
...
recurse(P[illegible]e, ...) {
   if (truncate?(p, node, ...))
      { ... }
   recurse(p, node.child1, ...);
   recurse(p, node.child2, ...);
   ...
}
```

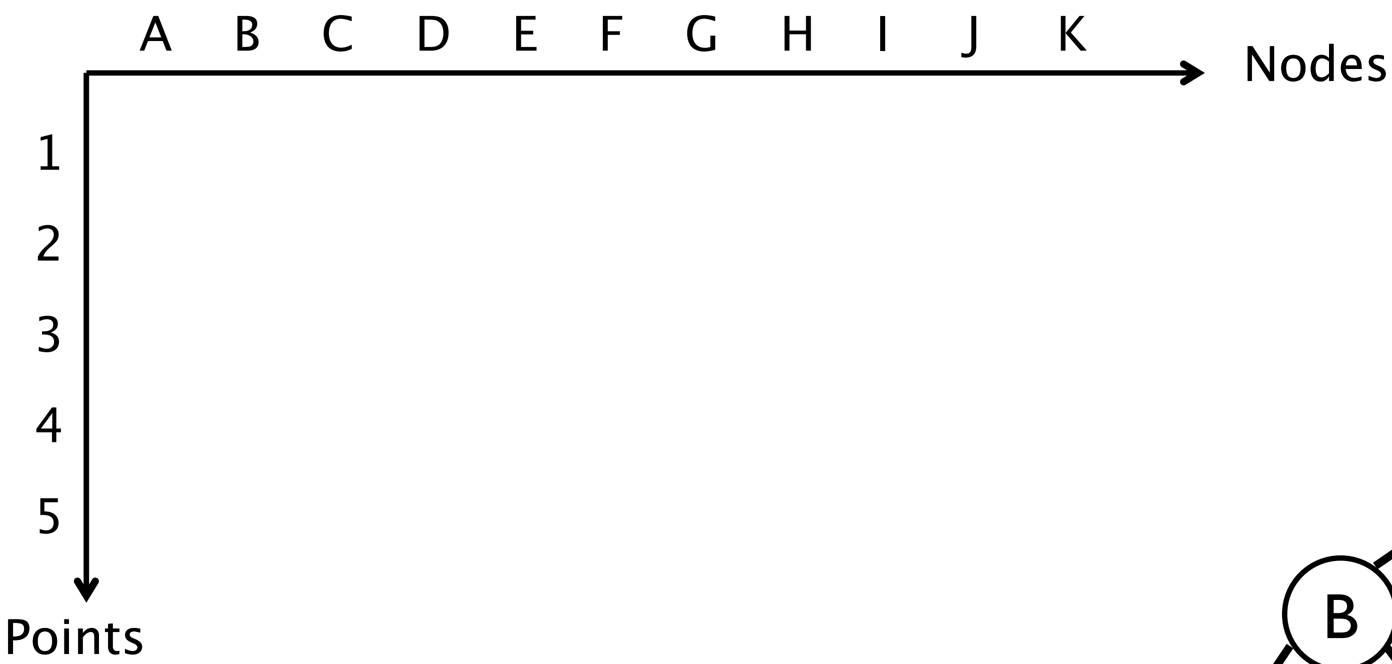
Let's ignore it!

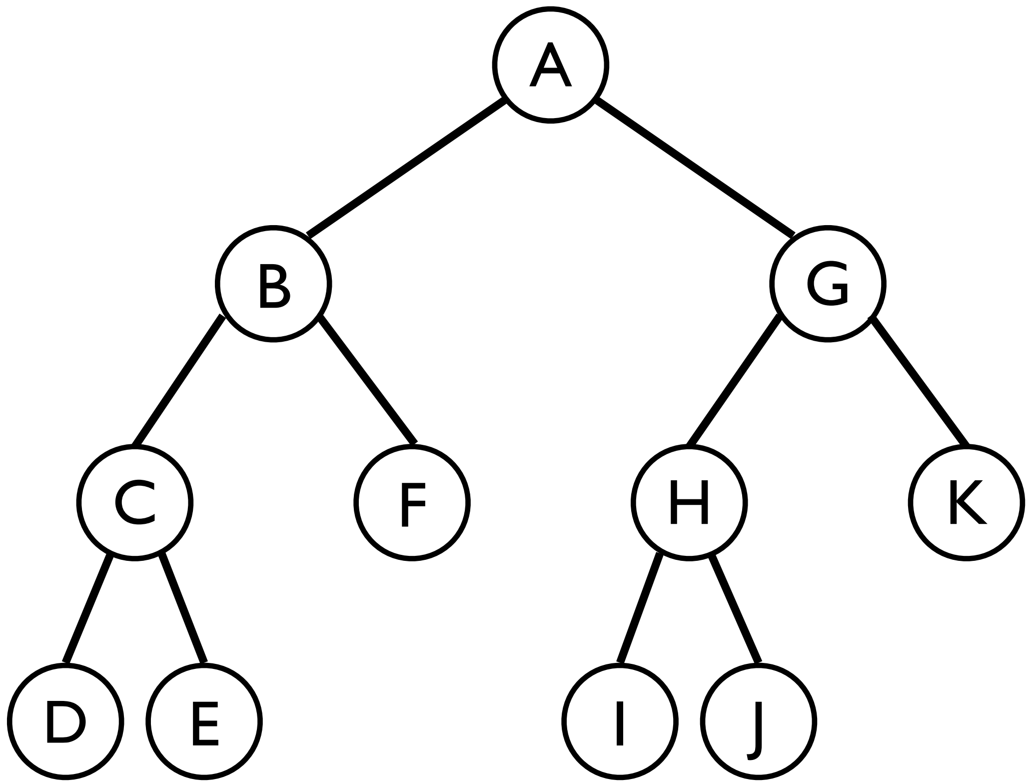
An abstract model

- Irregular traversals are tricky due to all the pointer chasing and recursion.
- Imagine there is an oracle that tells us which nodes we must traverse:

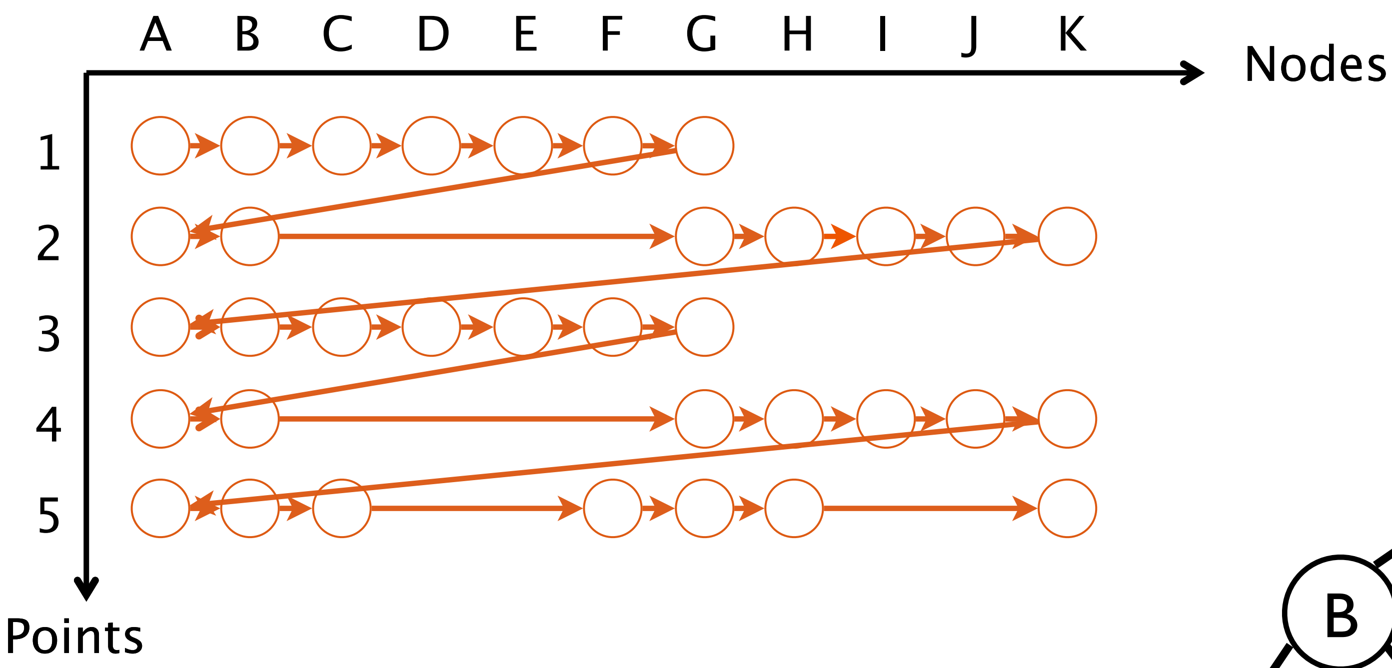
```
foreach Point p in ps {
   foreach TreeNode t in oracleTraverse(p) {
      interact(p, t);
   }
}
```

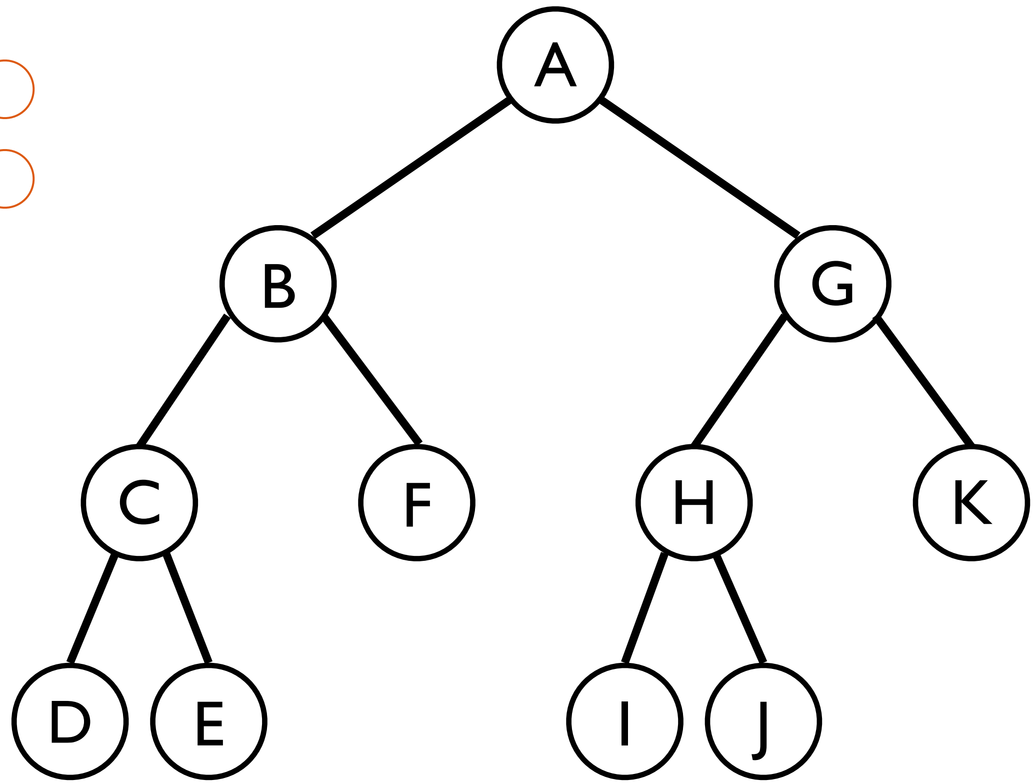
Reasoning about locality



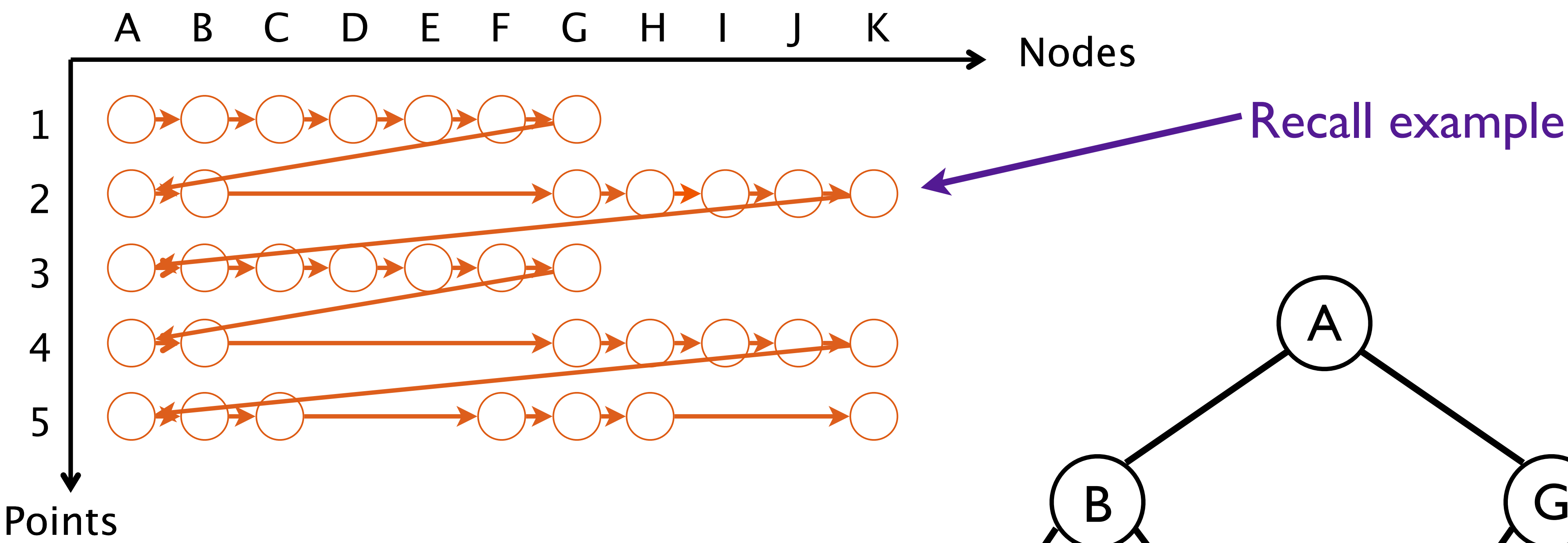


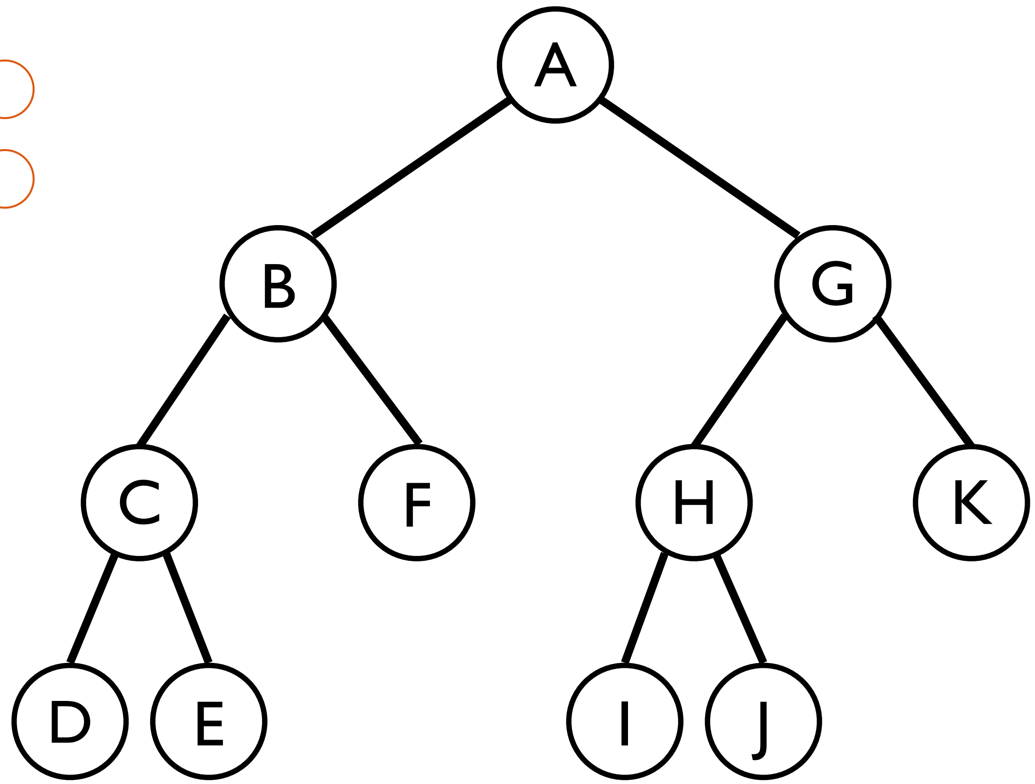
Reasoning about locality



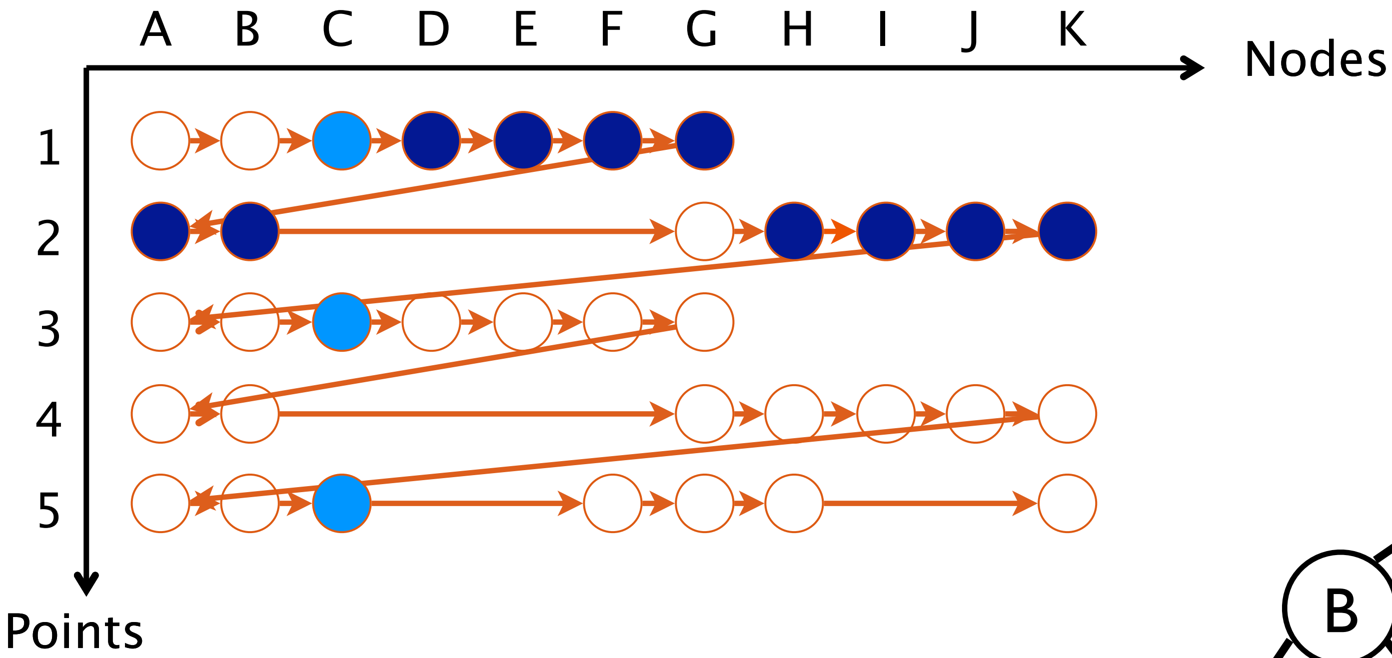


Reasoning about locality

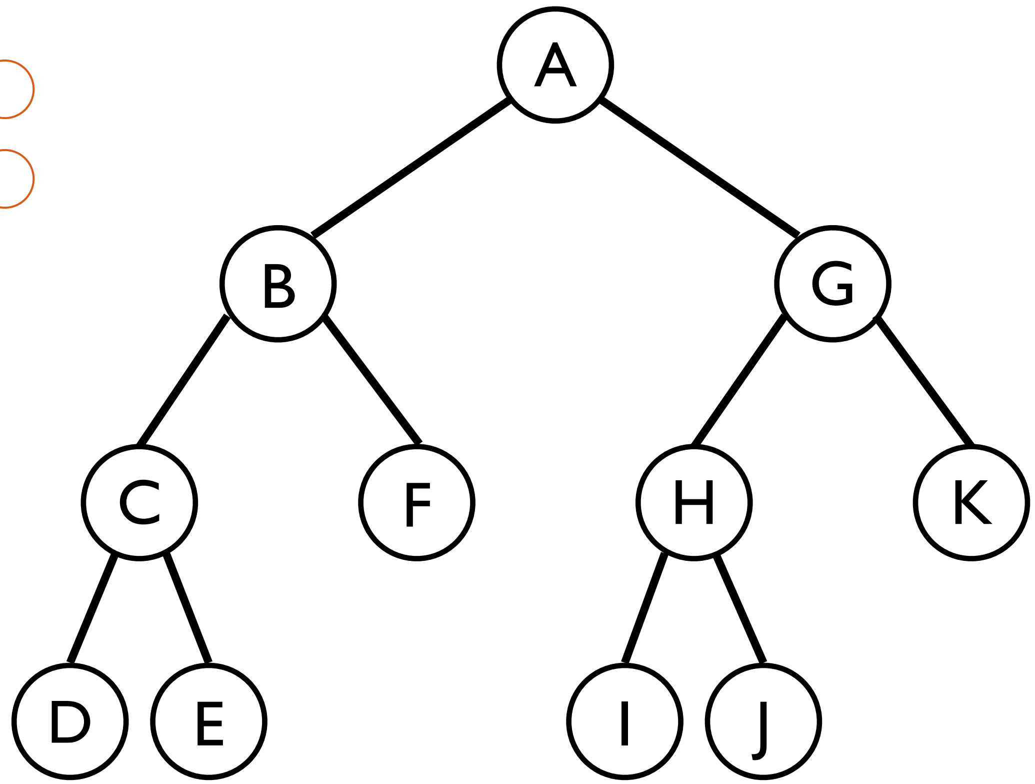




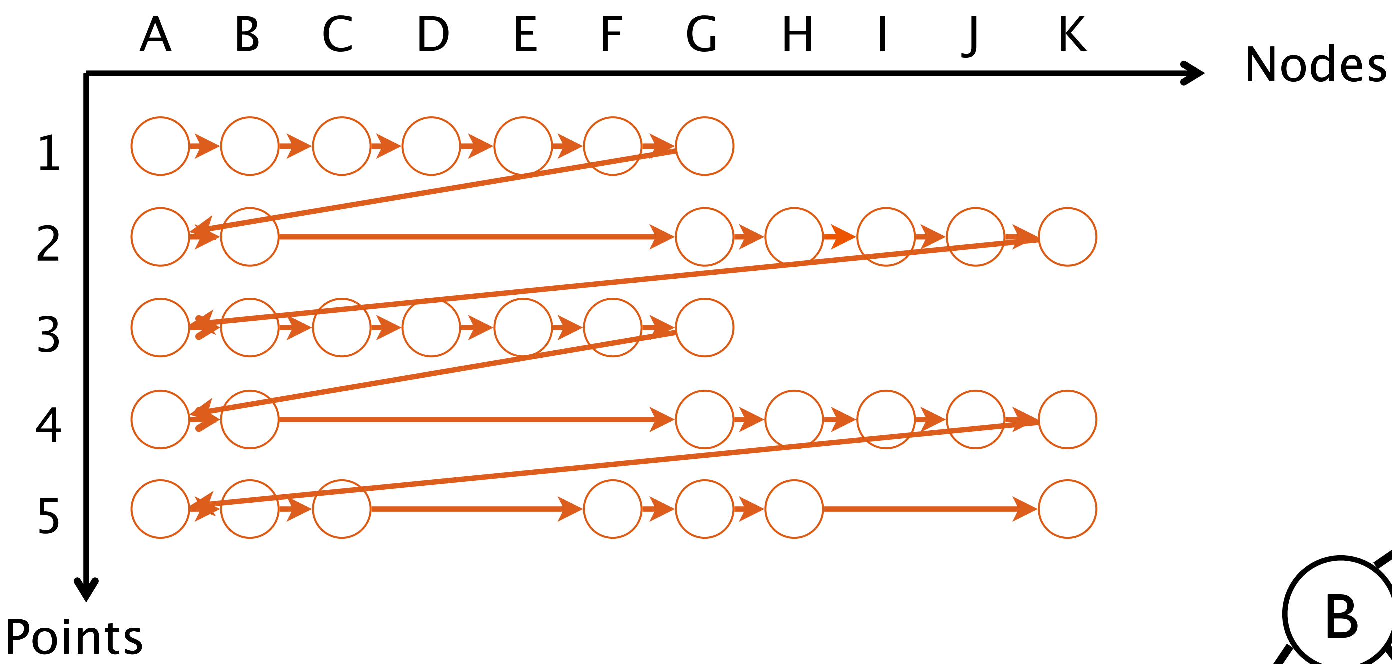
Reasoning about locality



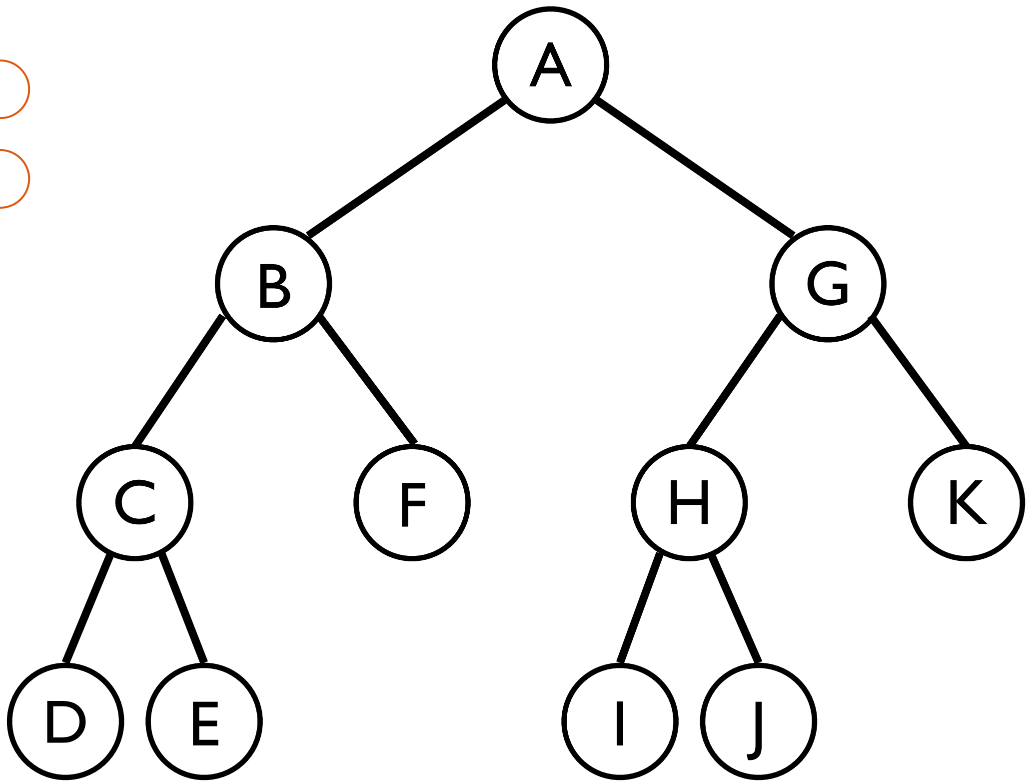
Reuse distance: 10



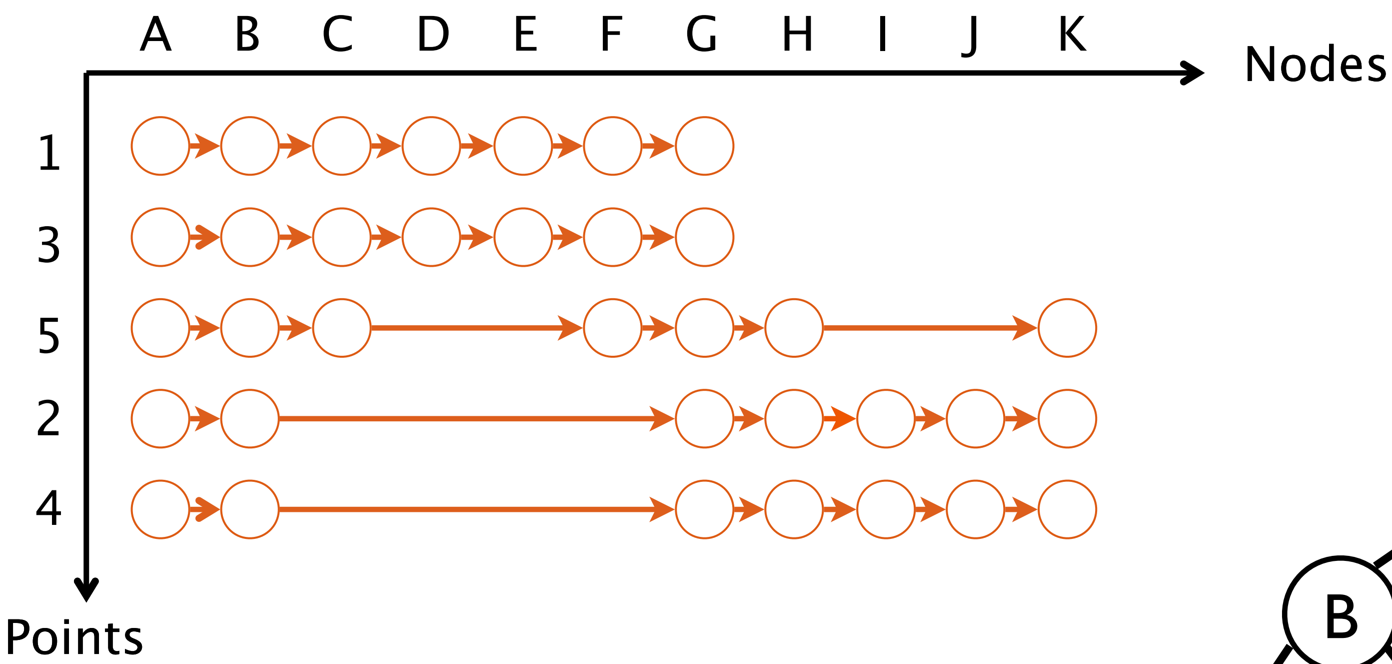
Reasoning about locality



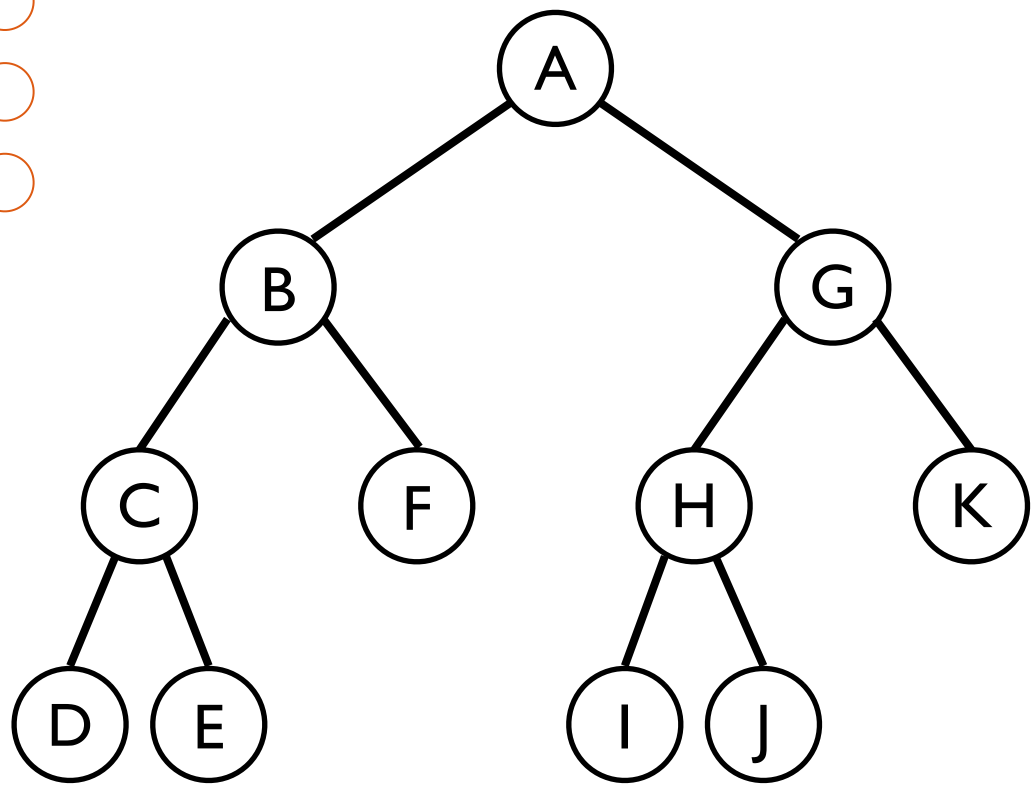
Reorder points to improve traversal overlap [Singh et al. 95, Amor et al. 00]



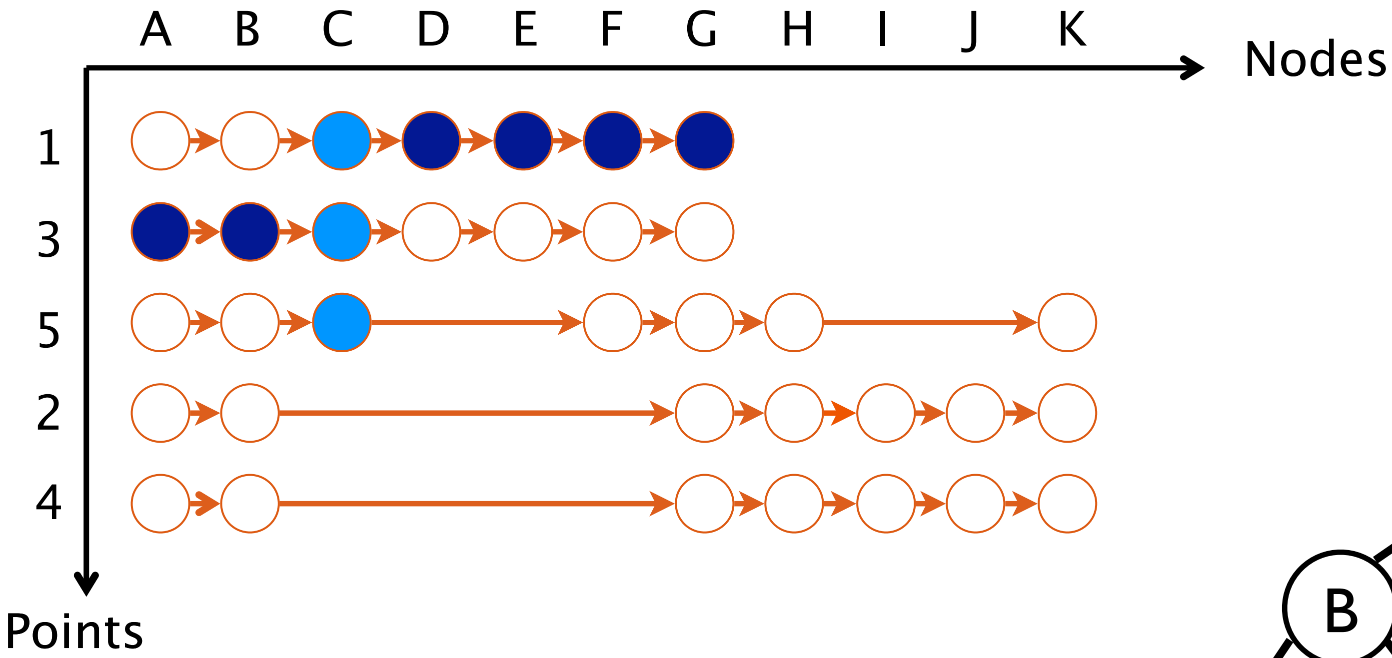
Reasoning about locality

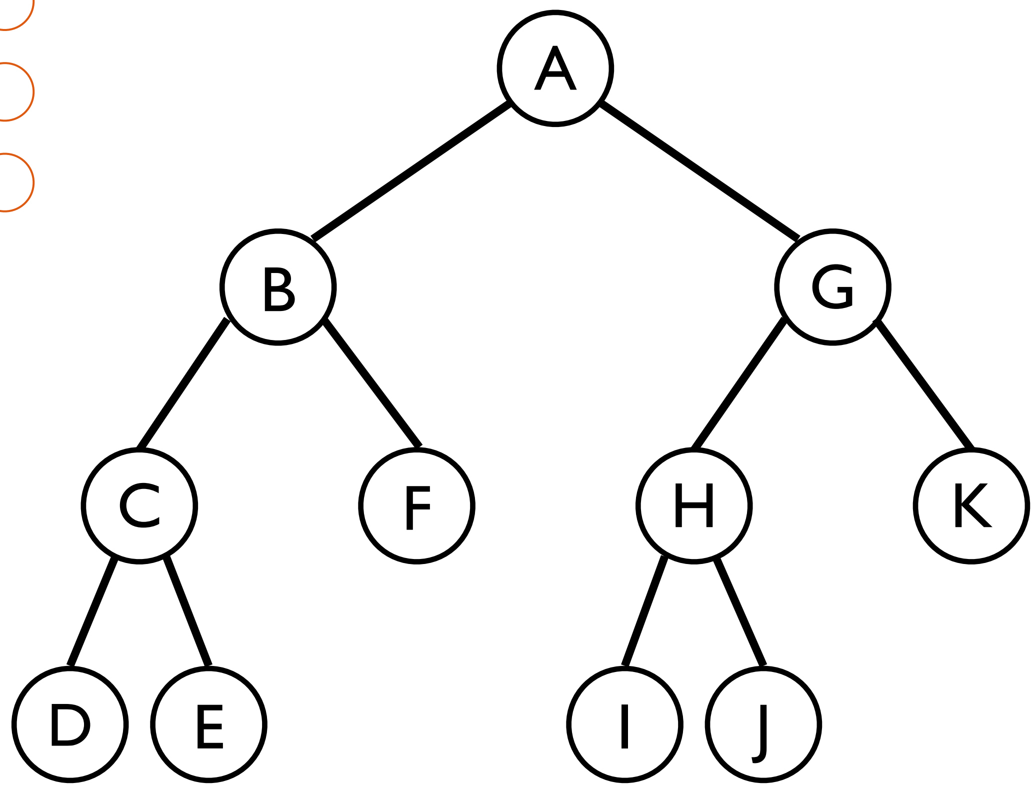


Reorder points to improve traversal overlap [Singh et al. 95, Amor et al. 00]



Reasoning about locality

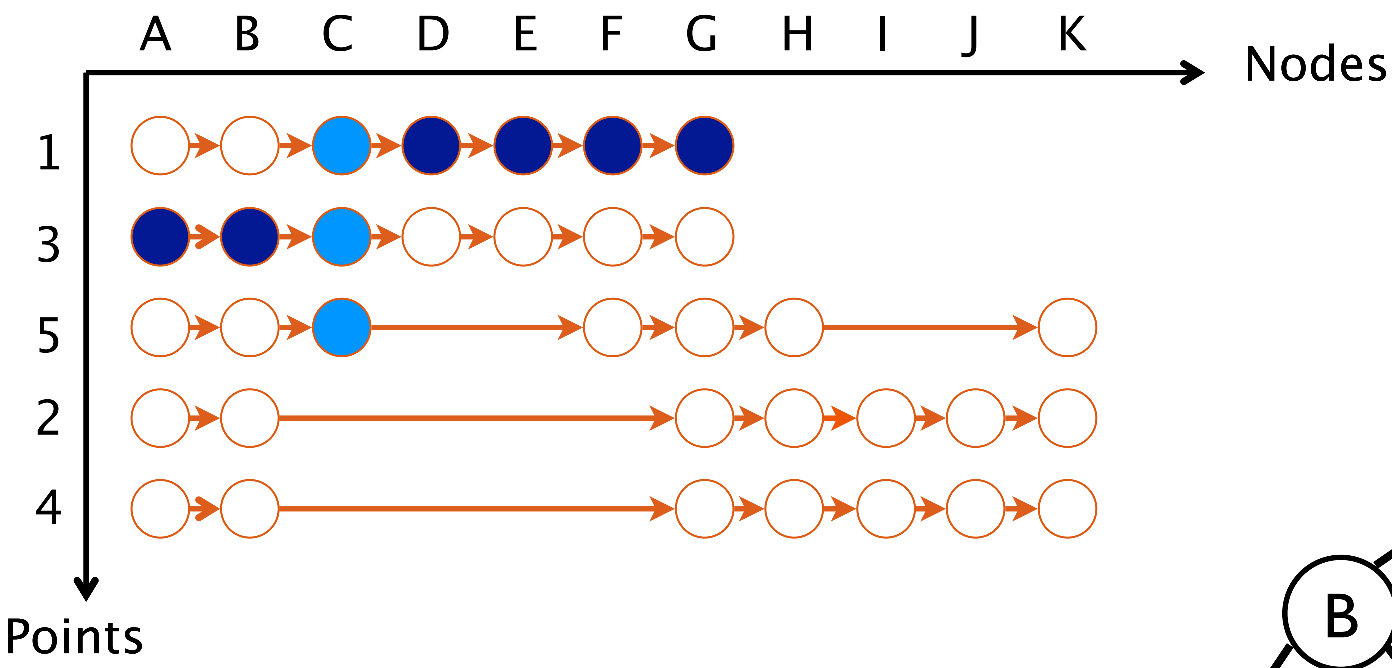


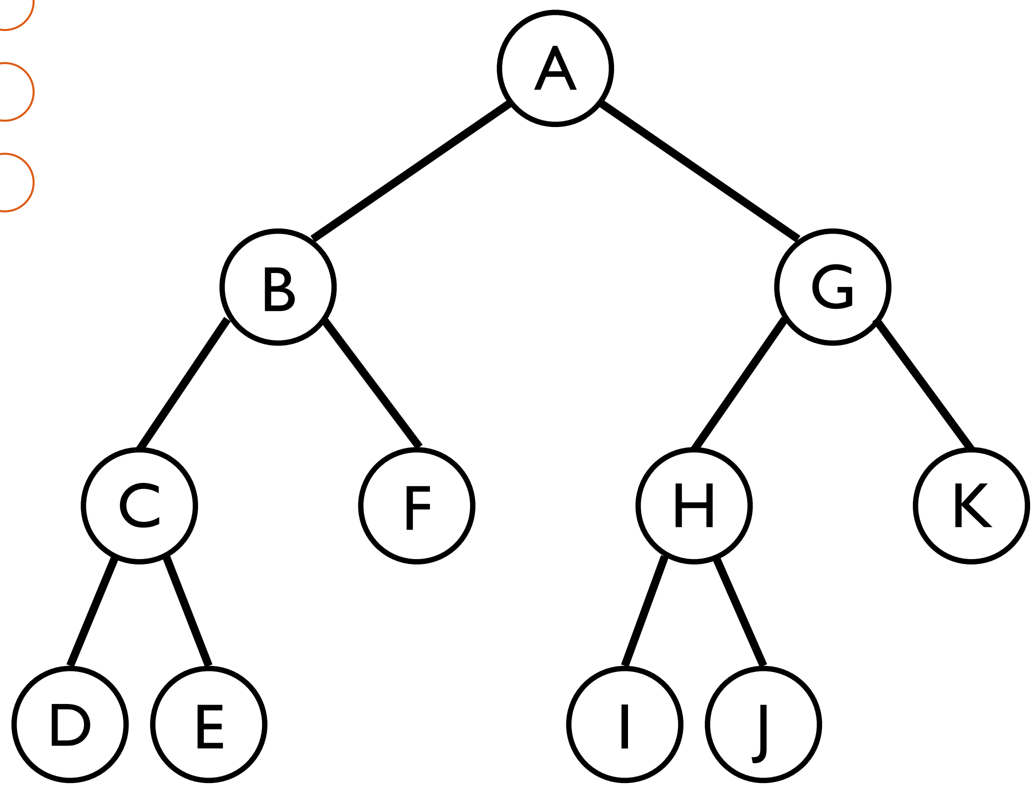


Reorder points to improve traversal overlap [Singh et al. 95, Amor et al. 00]

Reuse distance: 6

Reasoning about locality





So what happens when traversals get larger? (Alternate: caches get smaller)

→ Worst-case behavior!

Sorting behavior (Barnes-Hut)

# of bodies	Avg. Traversal (bytes)	L2 miss rate (%)	Improvement in cycles over unsorted (%)
10,000	63,944	21.61	67.3
100,000	108,656	44.97	45.9
1,000,000	139,616	55.30	26.4

Refining the model

- When the points are sorted, the difference between consecutive traversals is a second order effect.
    - Let's ignore that, too

```
foreach Point p in ps {
   foreach TreeNode t in oracleTraverse(p) {
      interact(p, t);
   }
}
```

Refining the model

- When the points are sorted, the difference between consecutive traversals is a second order effect.
  - Let's ignore that, too

```
foreach Point p in ps {
   foreach TreeNode t in ts {
      interact(p, t);
   }
}
```

Refining the model

- This has the same temporal locality behavior as vector outer-product:

```
foreach Point p in ps {
   foreach TreeNode t in ts {
      interact(p, t);
   }
}
```

Refining the model

- This has the same temporal locality behavior as vector outer-product:

```
for (i = 0; i < ps.size; i++) {
   for (j = 0; j < ts.size; j++) {
      interact(ps[i], ts[i]); //A[i][j] = ps[i]*ts[i]
   }
}
```

Refining the model

- This has the same temporal locality behavior as vector outer-product:

```
for (i = 0; i < ps.size; i++) {
    for (j = 0; j < ts.size; j++) {
        interact(ps[i], ts[i]); //A[i][j] = ps[i]*ts[i]
    }
}
```

Cold misses only

Refining the model

- This has the same temporal locality behavior as vector outer-product:

```
for (i = 0; i < ps.size; i++) {
    for (j = 0; j < ts.size; j++) {
        interact(ps[i], ts[i]); //A[i][j] = ps[i]*ts[i]
    }
}
```

Cold misses only (ps[i])

Capacity misses on each access (ts[i])

Loop transformations to the rescue!

- We can now reason about the application of classical loop transformations to traversal codes
  - e.g., interchange?

```
for (i = 0; i < ps.size; i++) {
   for (j = 0; j < ts.size; j++) {
      interact(ps[i], ts[i]);
   }
}
```

Loop transformations to the rescue!

- We can now reason about the application of classical loop transformations to traversal codes
  - e.g., interchange?

```
for (j = 0; j < ts.size; j++) {
   for (i = 0; i < ps.size; i++) {
      interact(ps[i], ts[i]);
   }
}
```

Loop transformations to the rescue!

- We can now reason about the application of classical loop transformations to traversal codes
  - e.g., interchange?

```
for (j = 0; j < ts.size; j++) {
   for (i = 0; i < ps.size; i++) {
      interact(ps[i], ts[i]);
   }
}
```

Capacity misses on each access

Loop transformations to the rescue!

- We can now reason about the application of classical loop transformations to traversal codes
  - e.g., interchange?

```
for (j = 0; j < ts.size; j++) {
    for (i = 0; i < ps.size; i++) {
        interact(ps[i], ts[i]);
    }
}
```

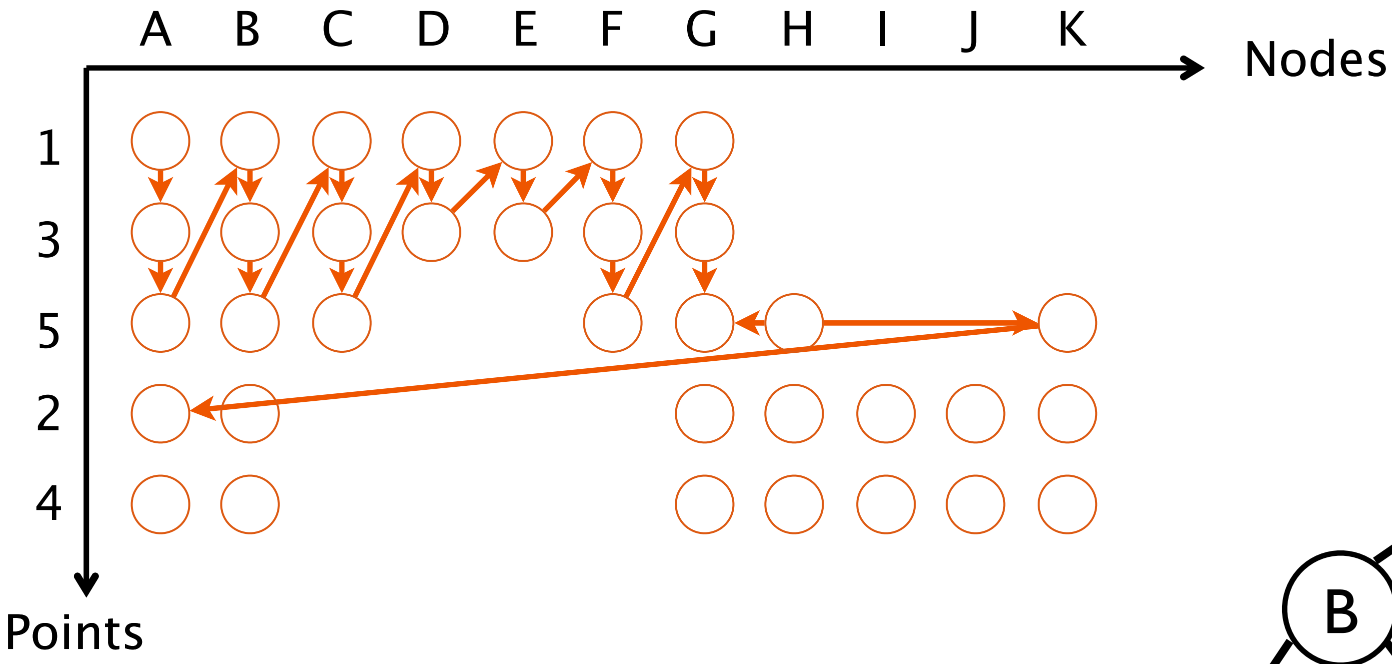
Capacity misses on each access

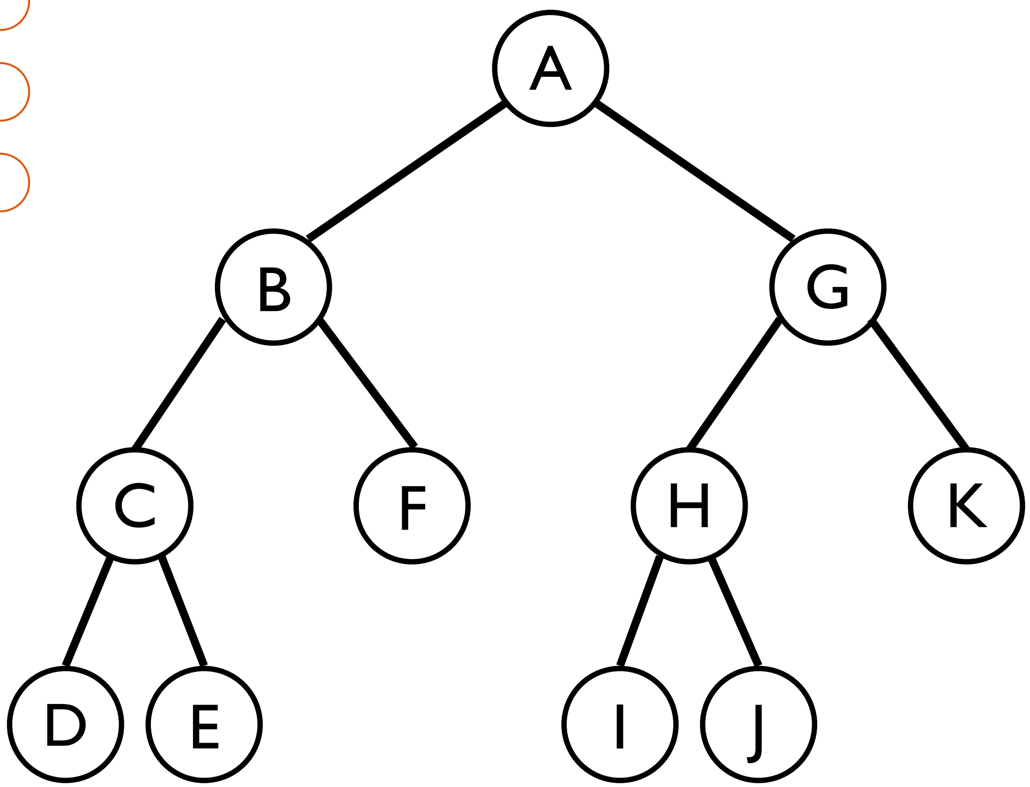
Cold misses only

Tiling

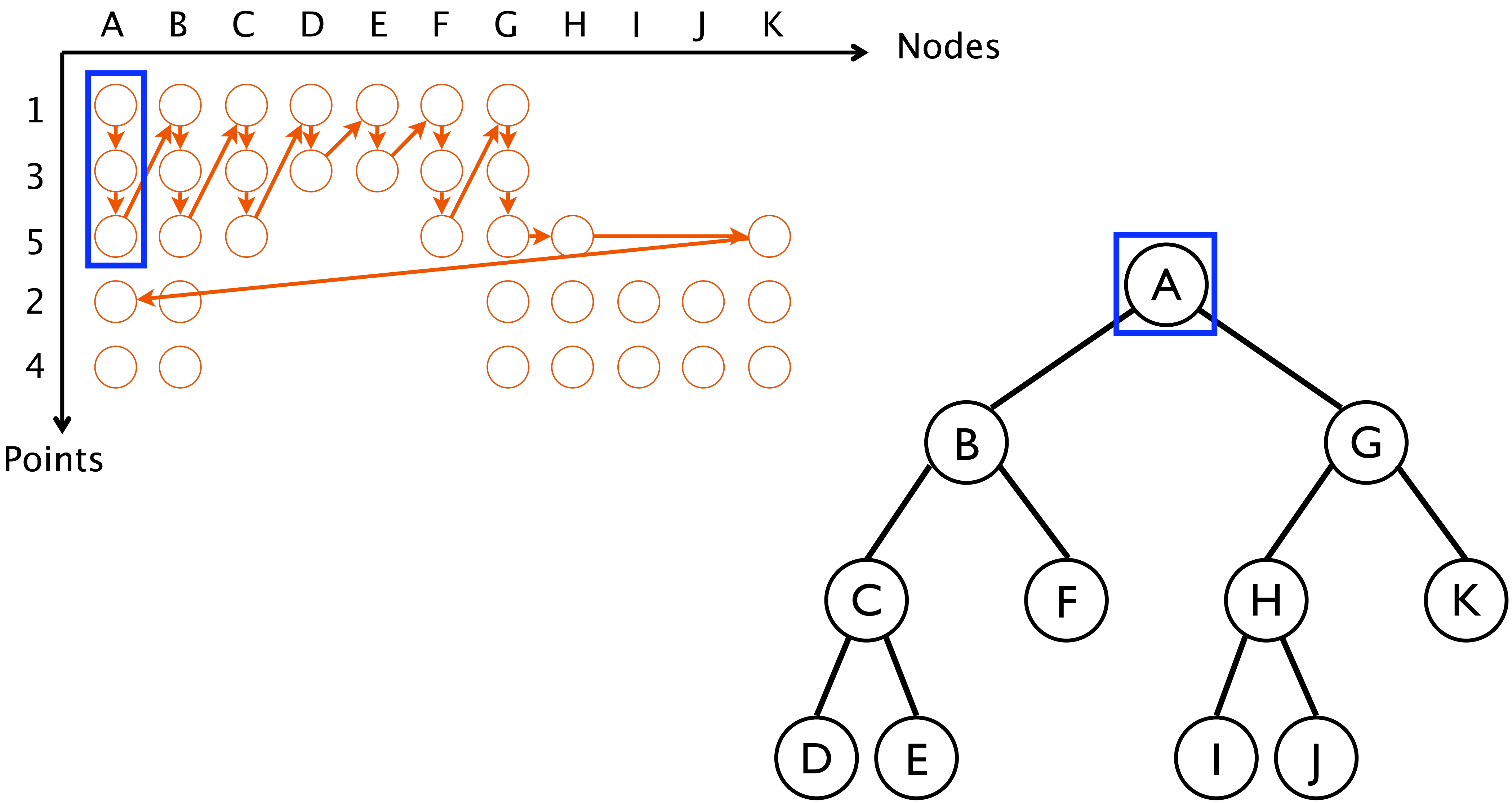
```
for (ii = 0; ii < ps.size; i += B) {
  for (j = 0; j < ts.size; j++) {
   for (i = ii; i < ii + B; i++) {
      interact(ps[i], ts[i]);
    }
  }
}

foreach Block<Point> bp in ps {
   foreach TreeNode t in oracleTraverse(bp) {
      foreach Point p in bp {
         interact(p, t);
      }
   }
}
```

Point blocking



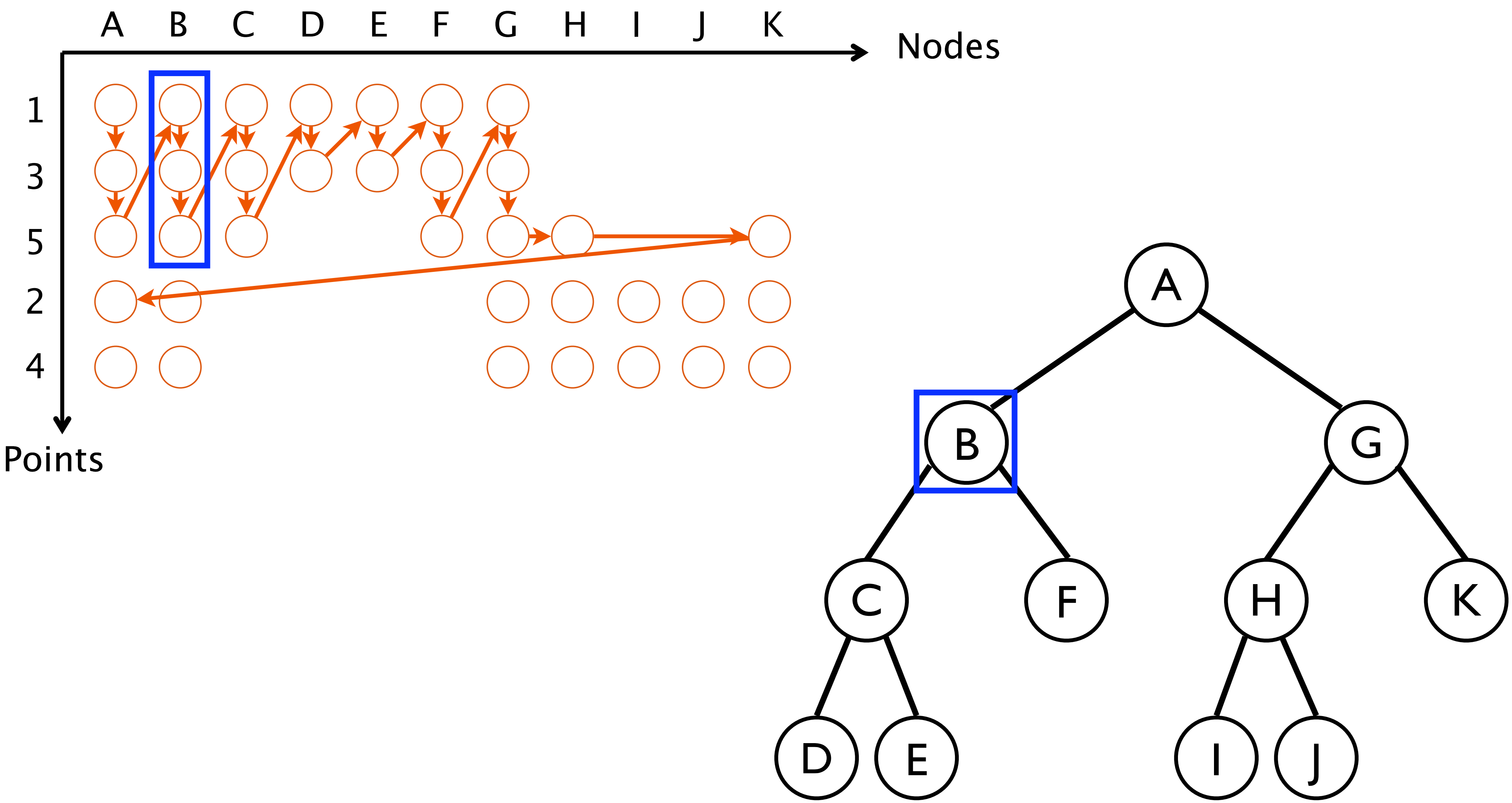


Point blocking

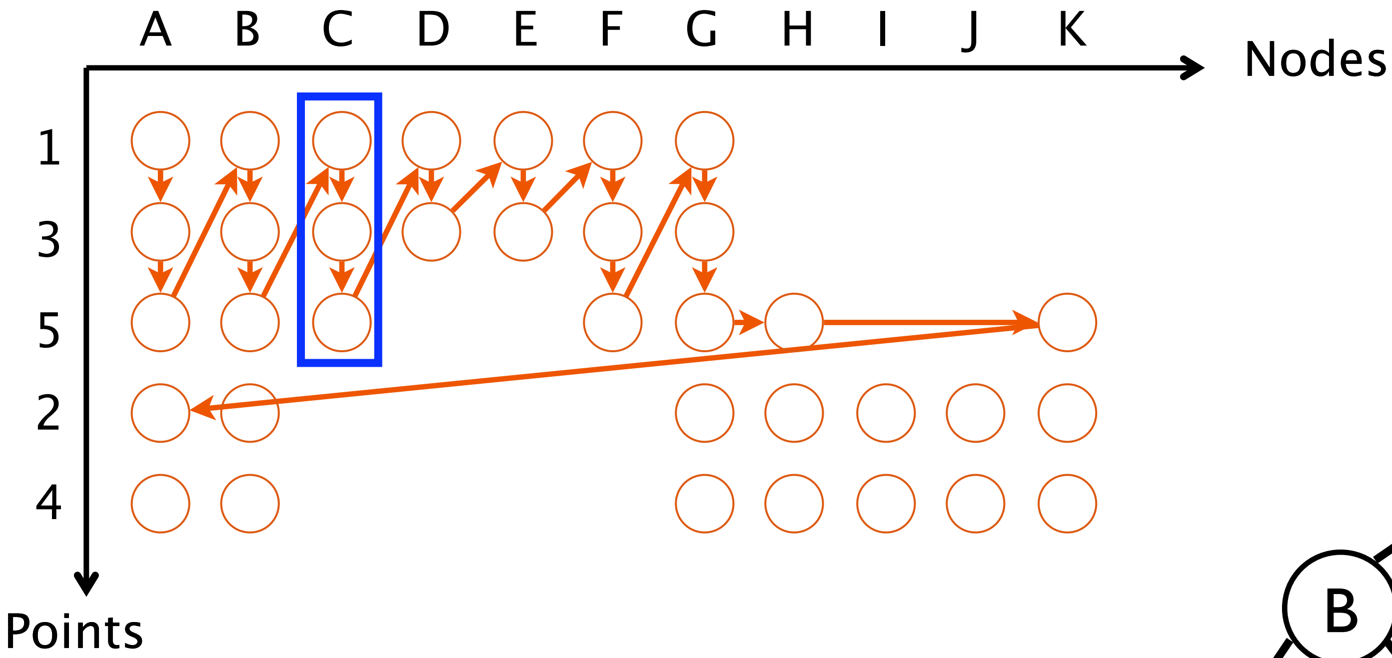


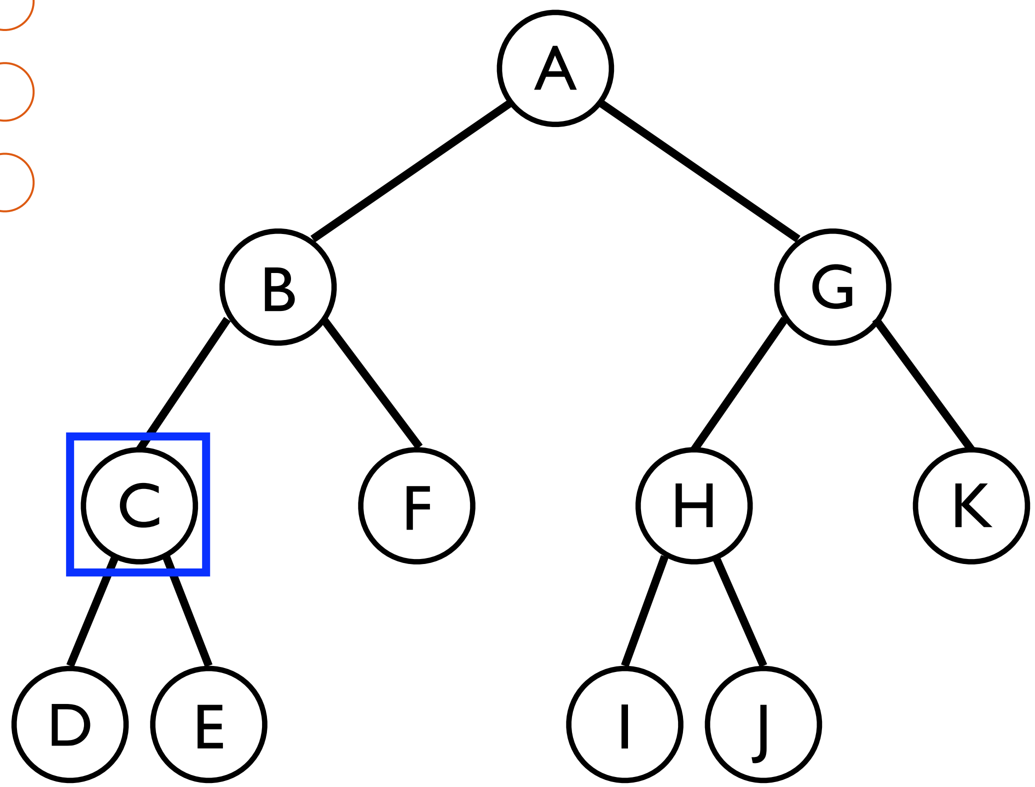
Tuesday, October 4, 2011

Point blocking

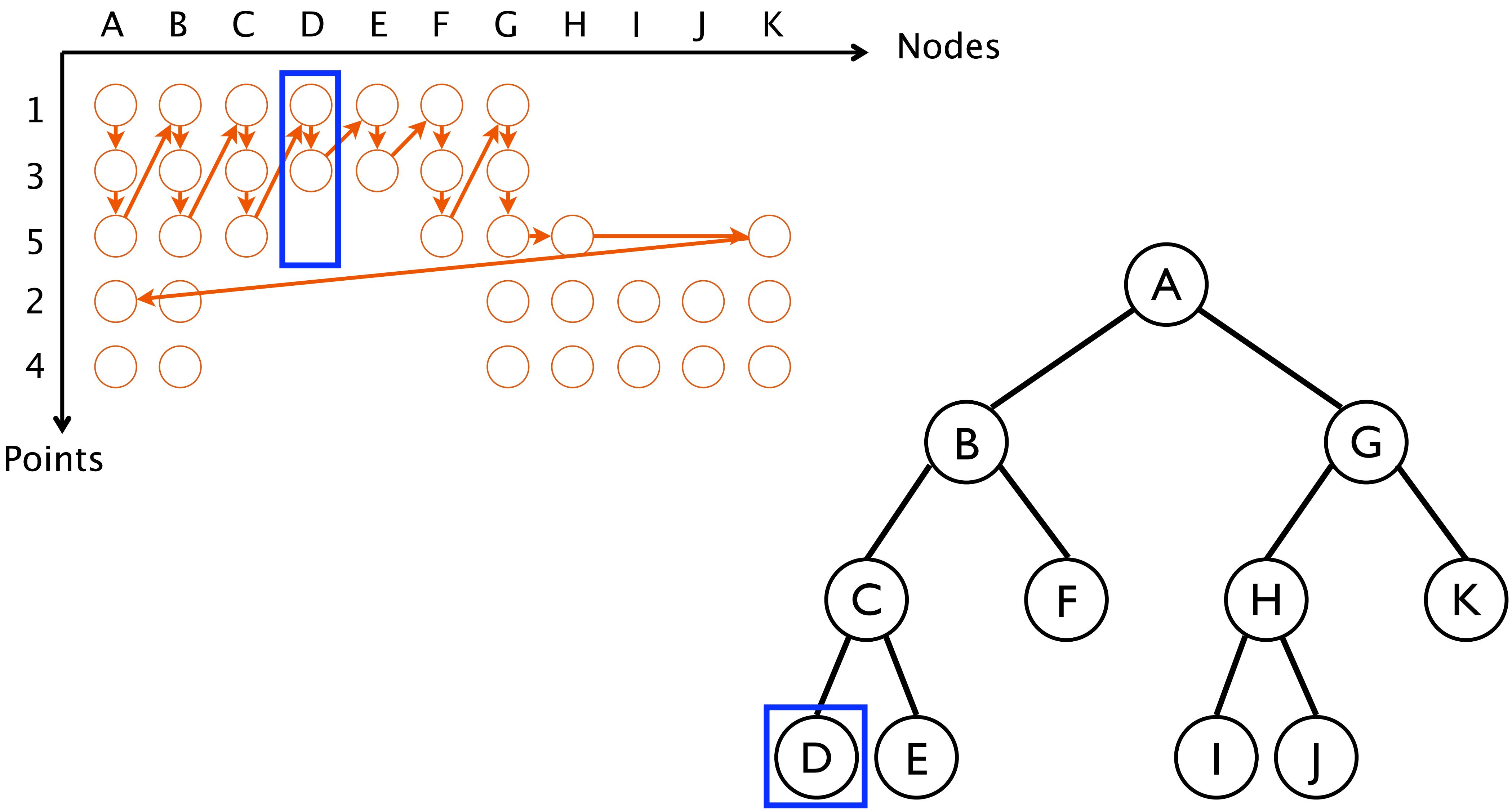


Point blocking

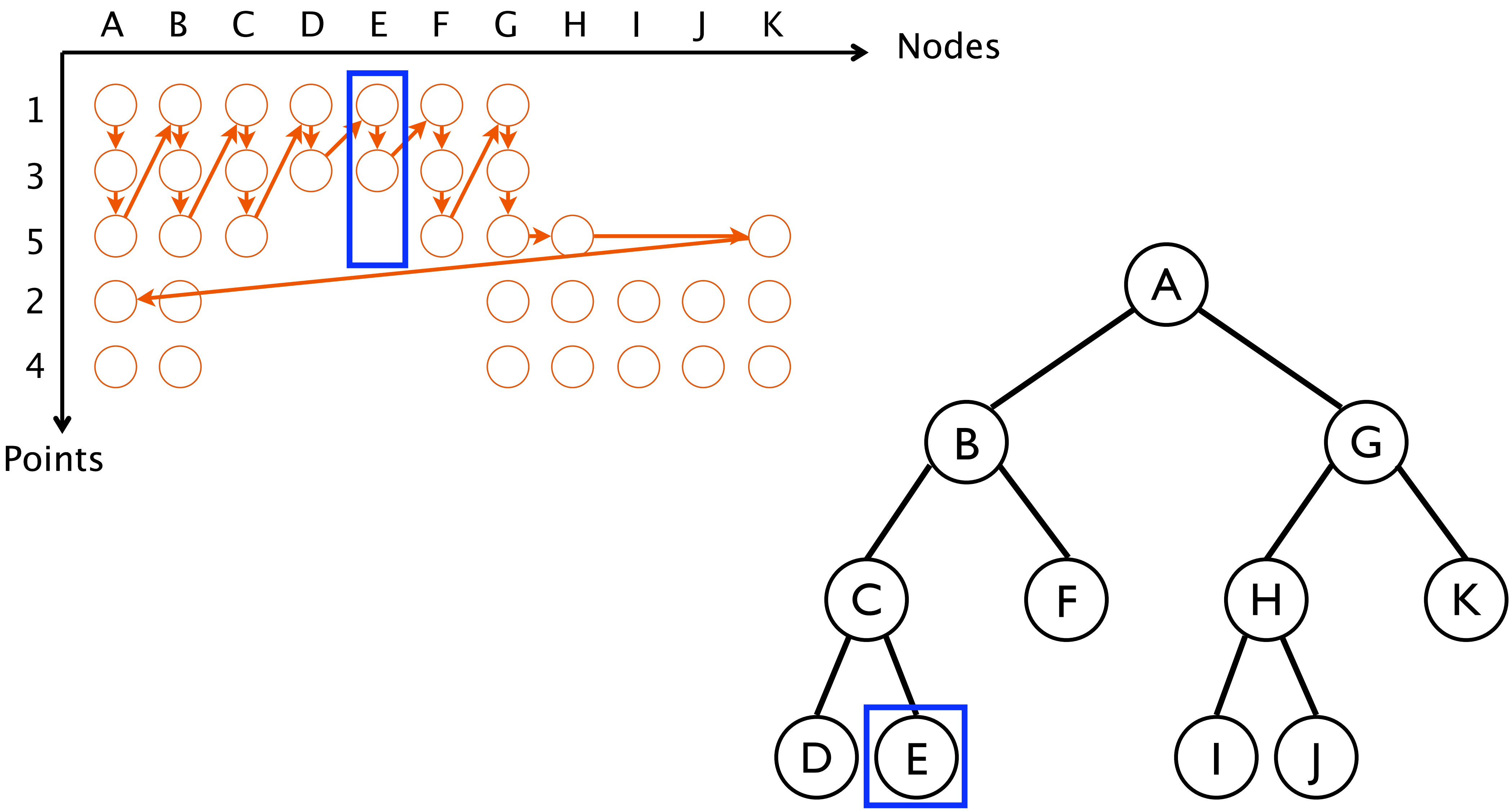




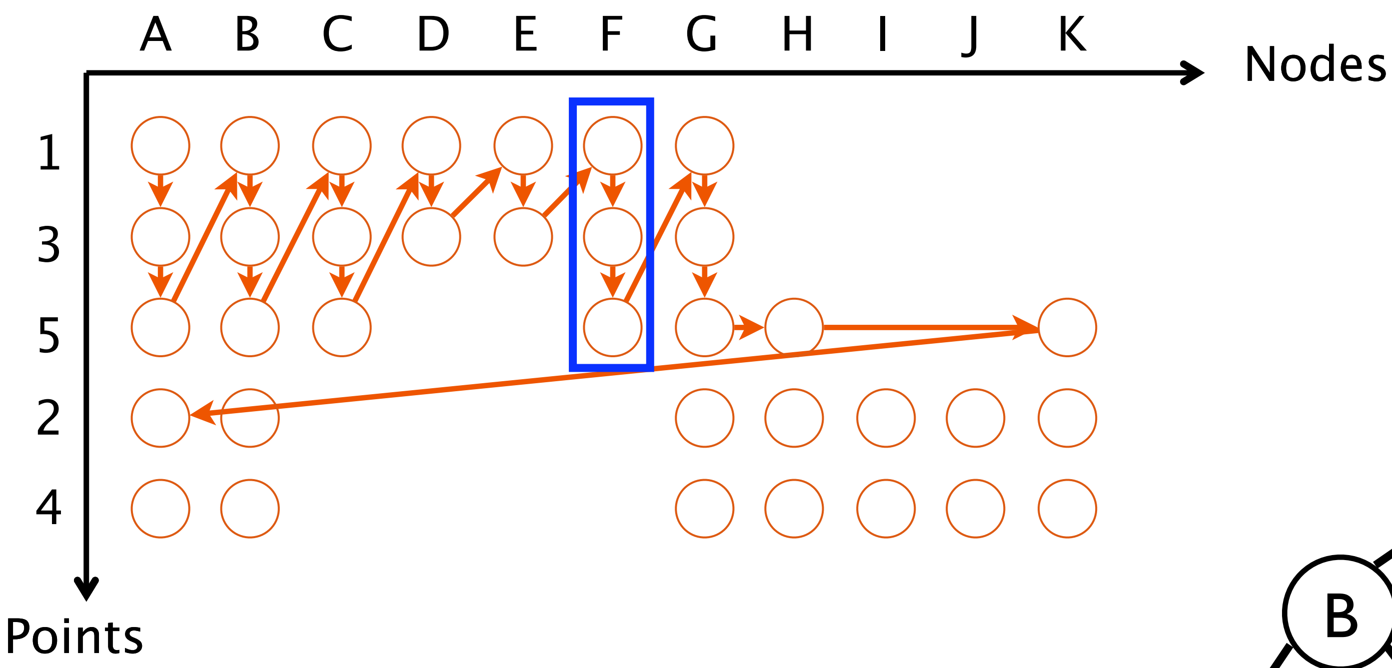
Point blocking

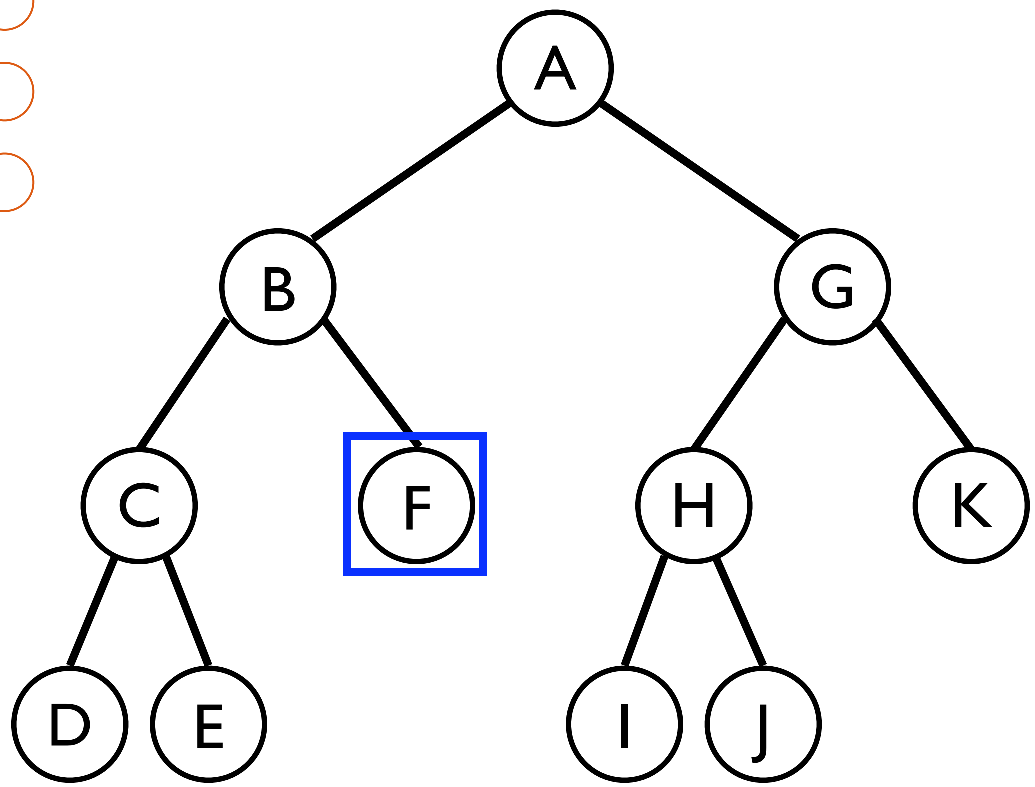


Point blocking

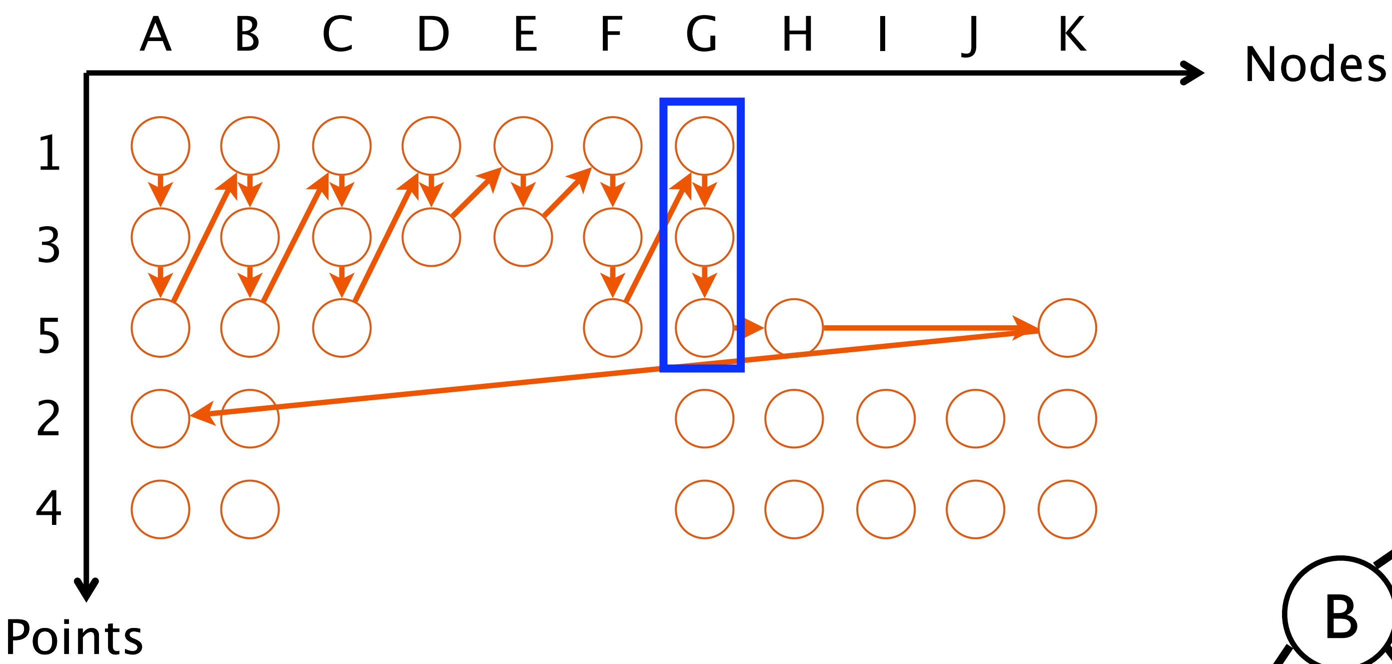


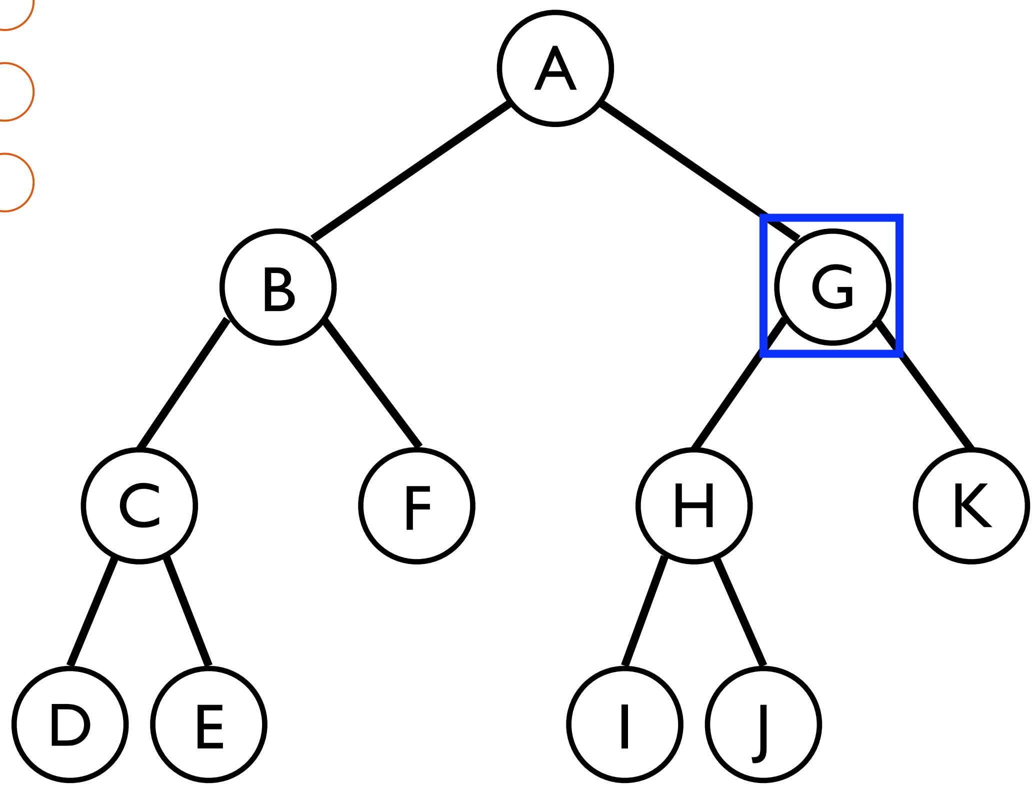
Point blocking



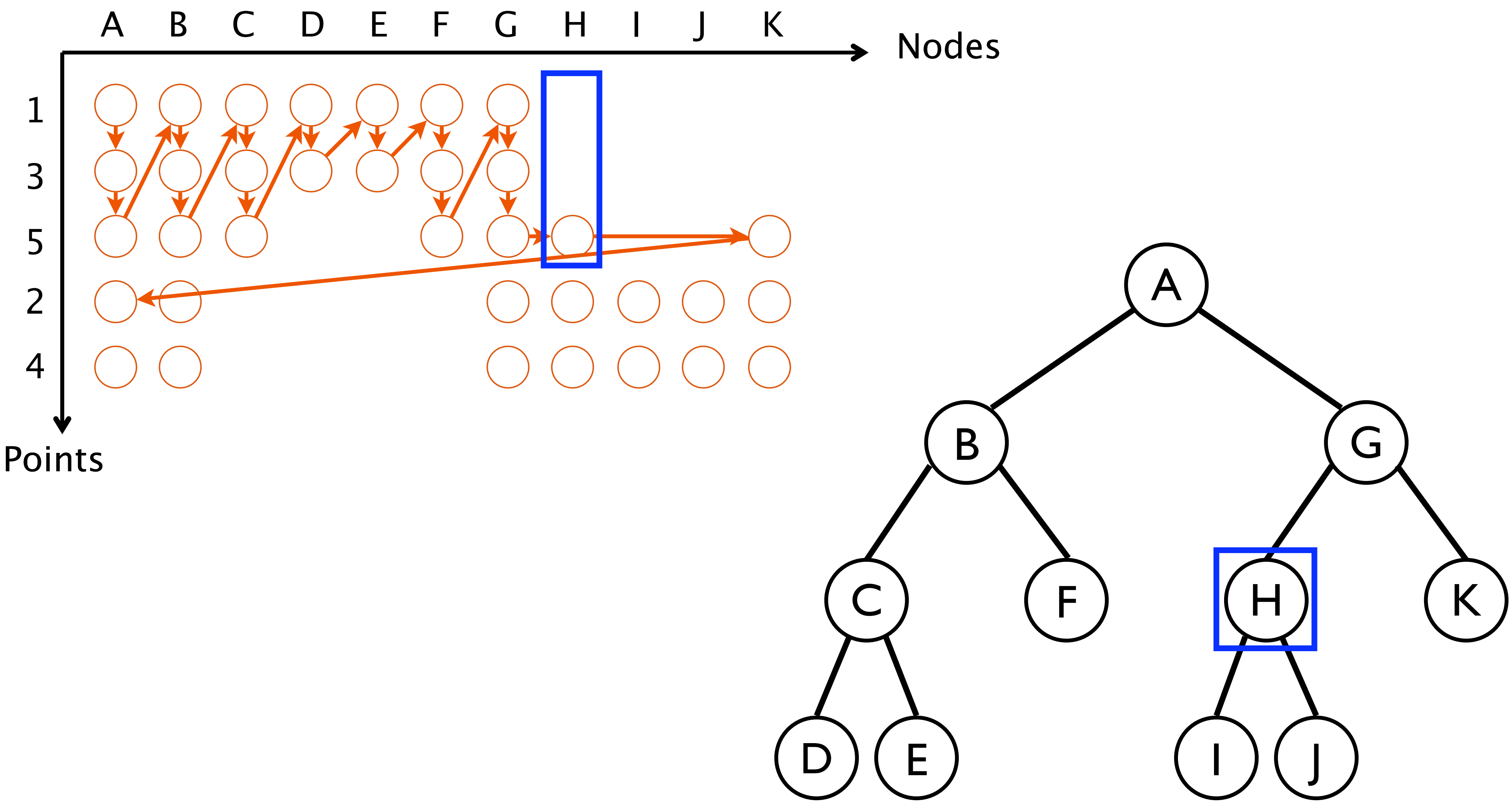


Point blocking

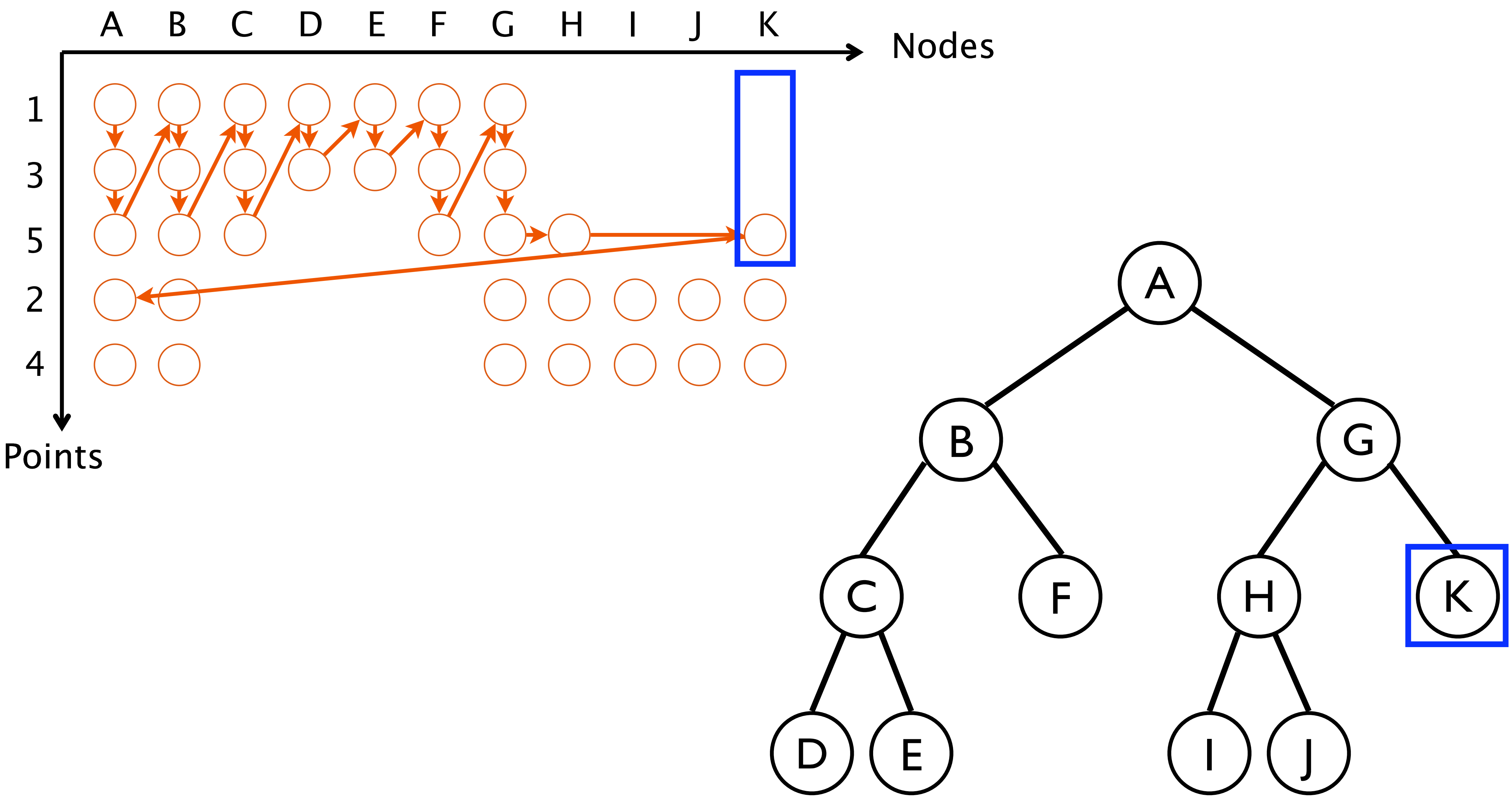




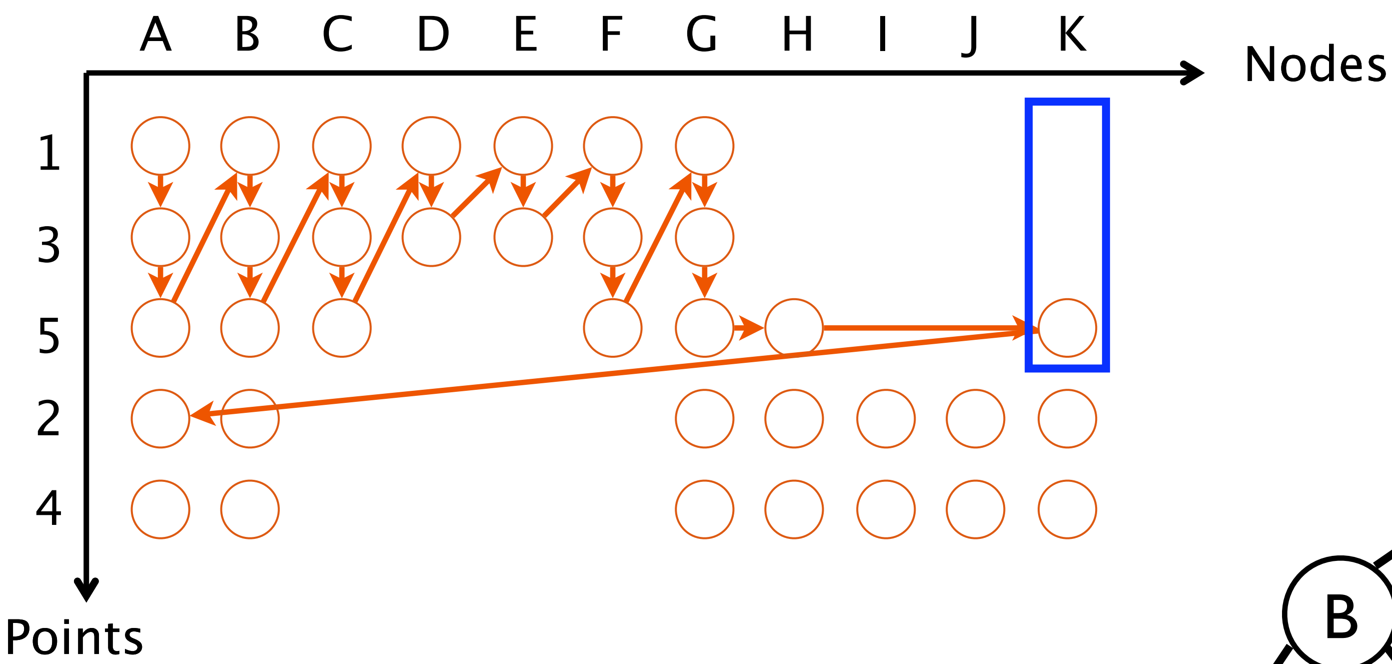
Point blocking

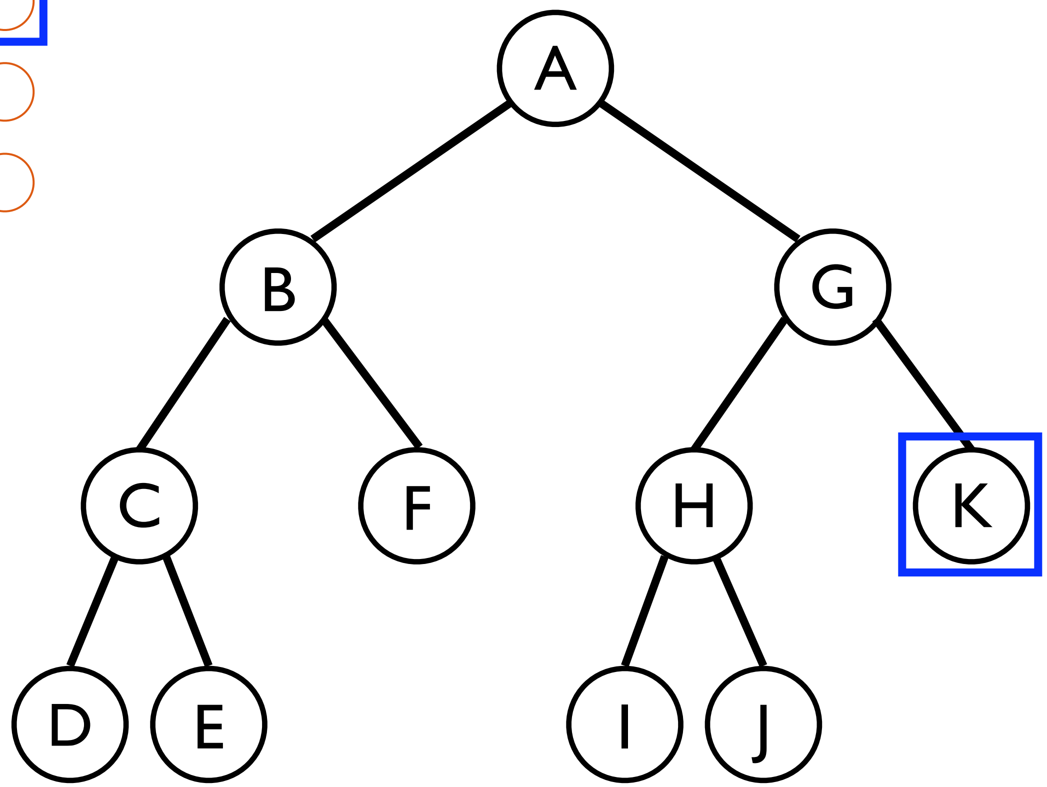


Point blocking



Point blocking

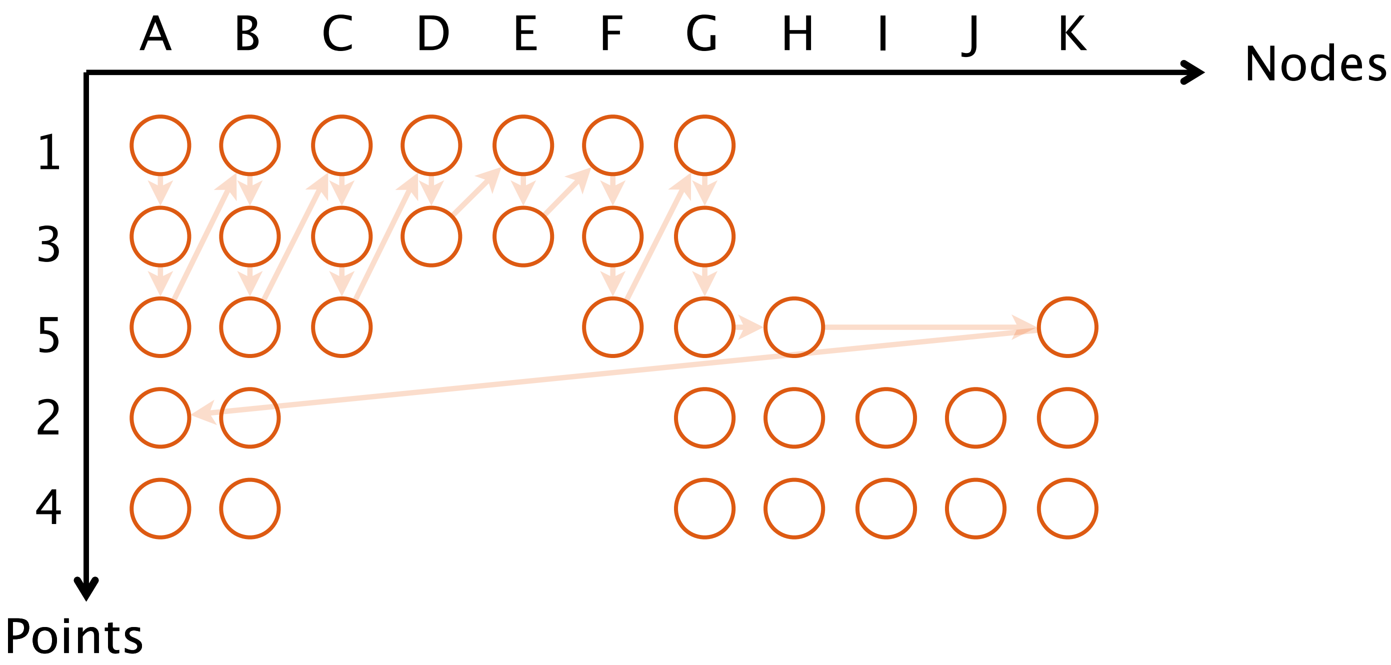




Locality intuition:

- Miss on each tree node once per block
- If block fits in cache, only cold misses on points

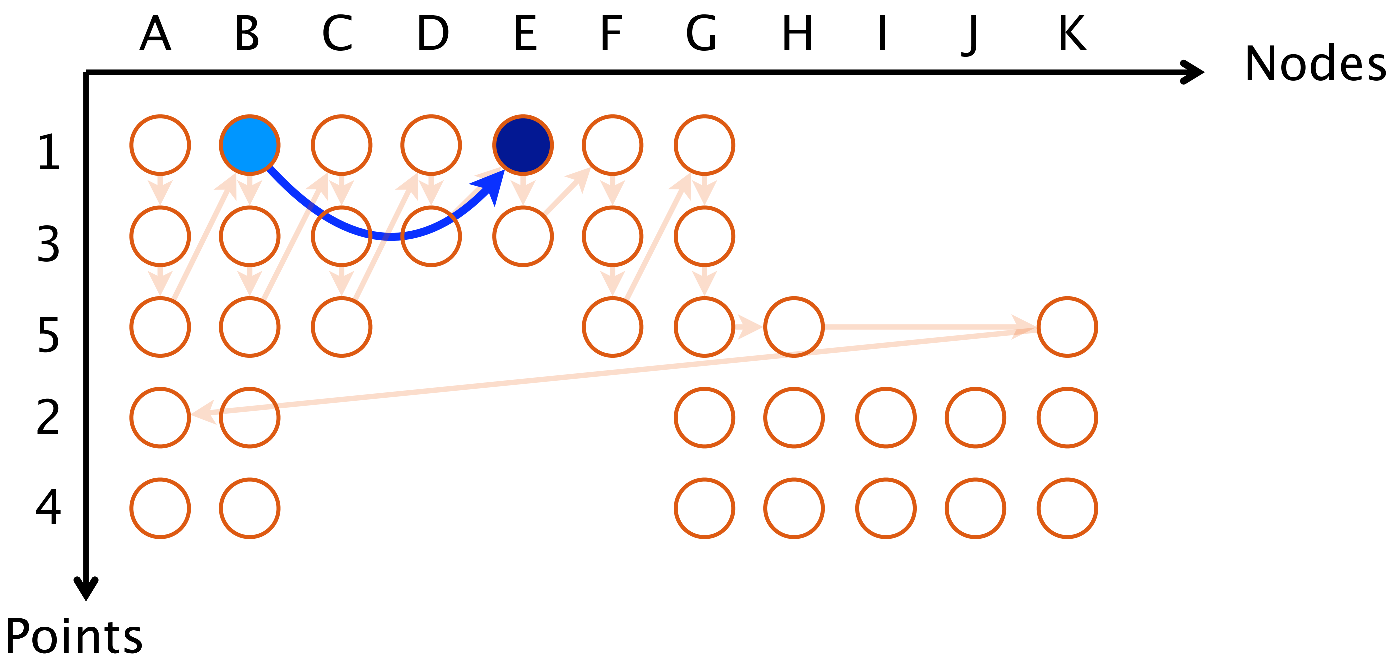
Reasoning about correctness



Dependence preserved by point blocking

Dependence violated by point blocking

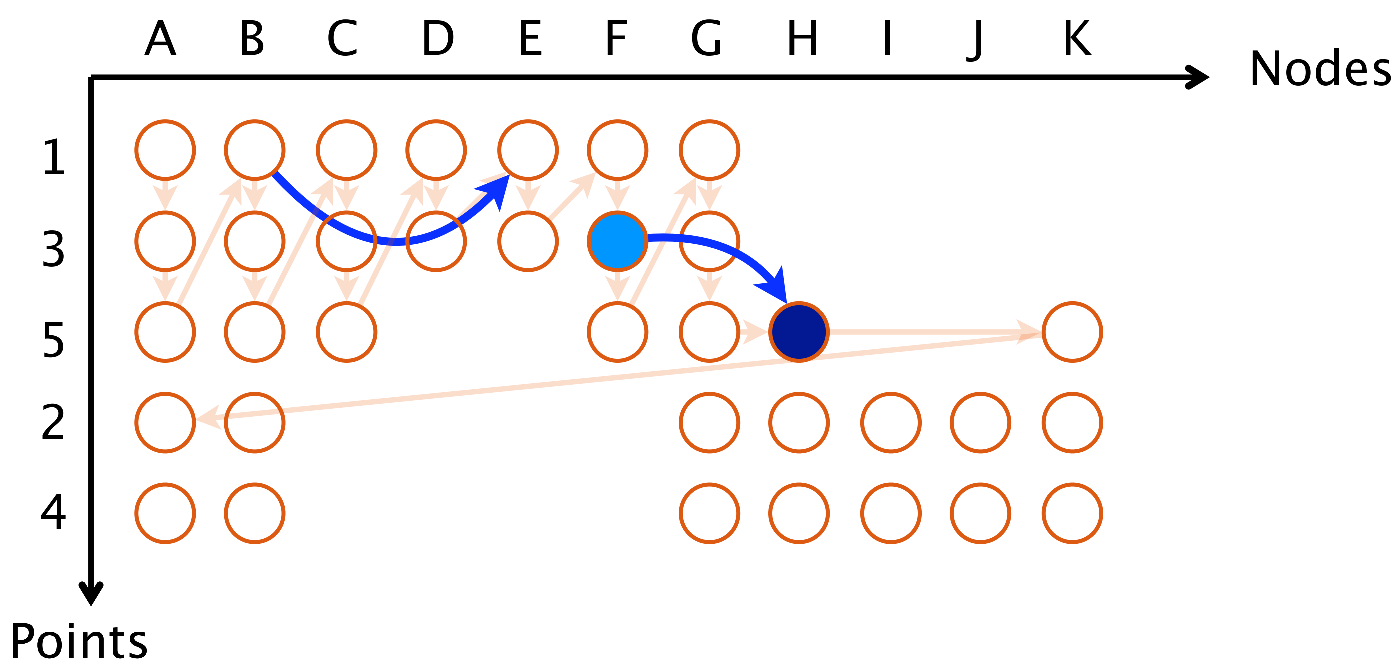
Reasoning about correctness



Dependence preserved by point blocking

Dependence violated by point blocking

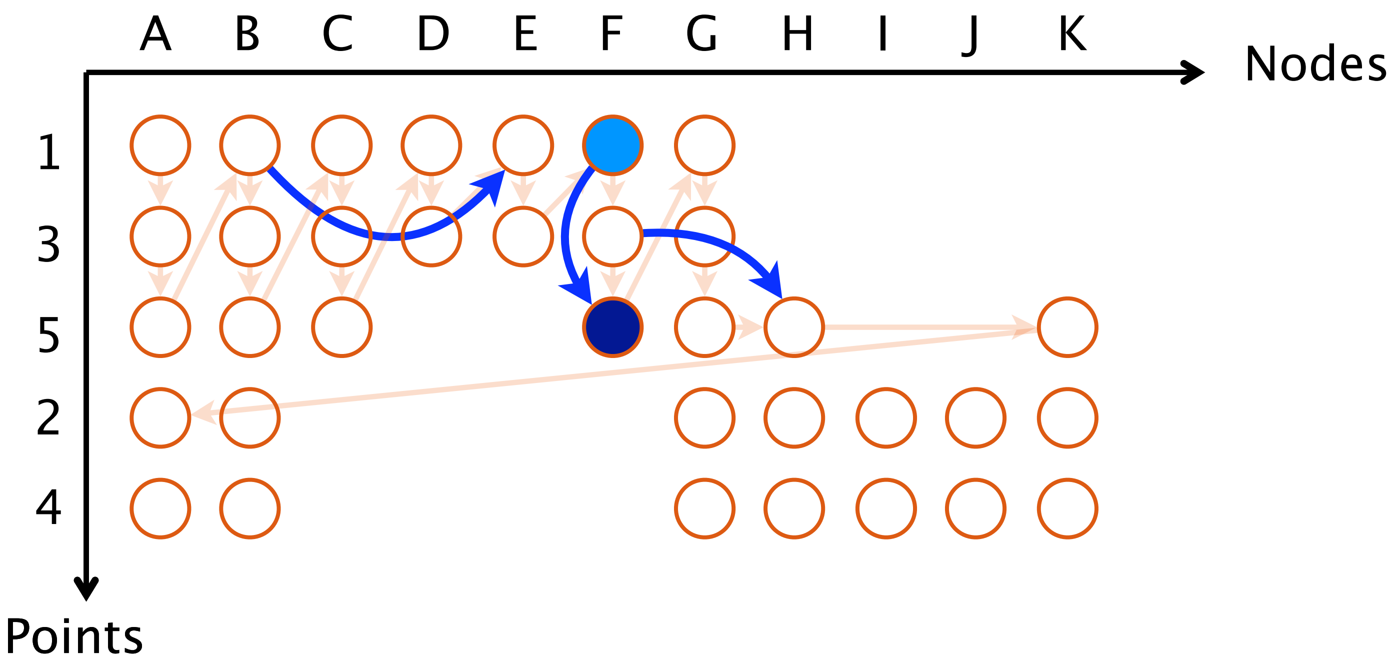
Reasoning about correctness



Dependence preserved by point blocking

Dependence violated by point blocking

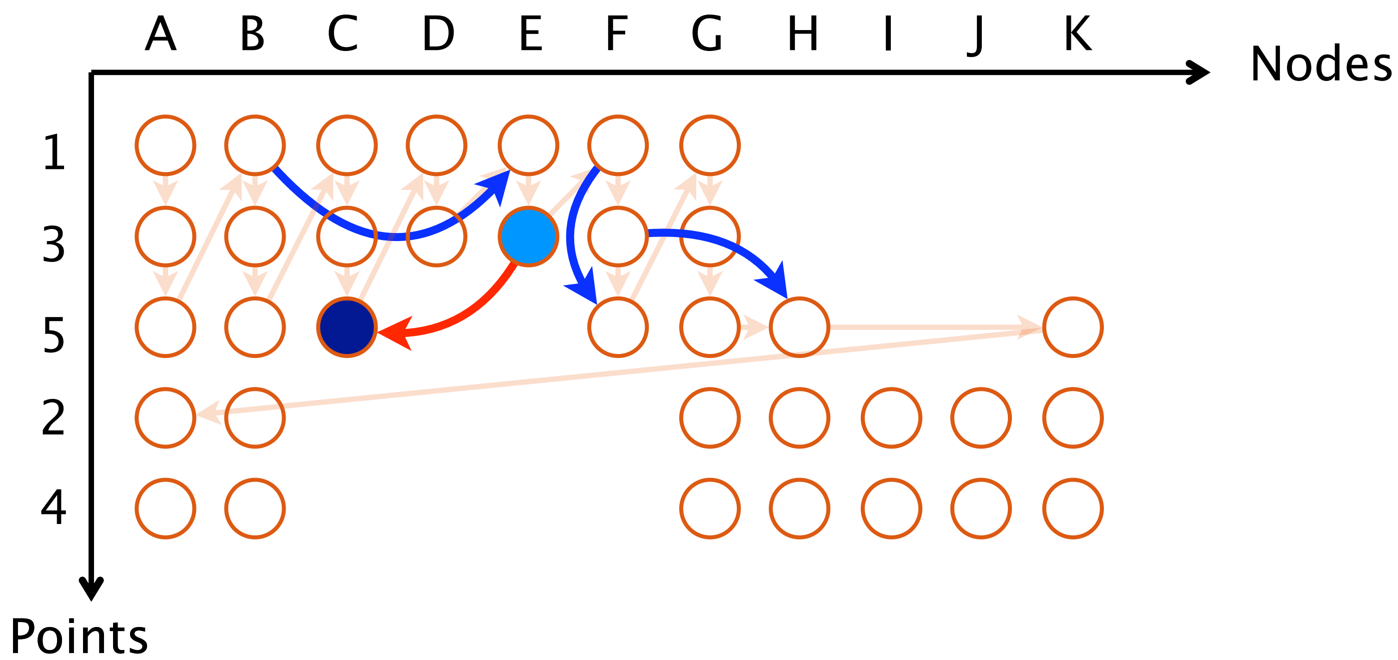
Reasoning about correctness



Dependence preserved by point blocking

Dependence violated by point blocking

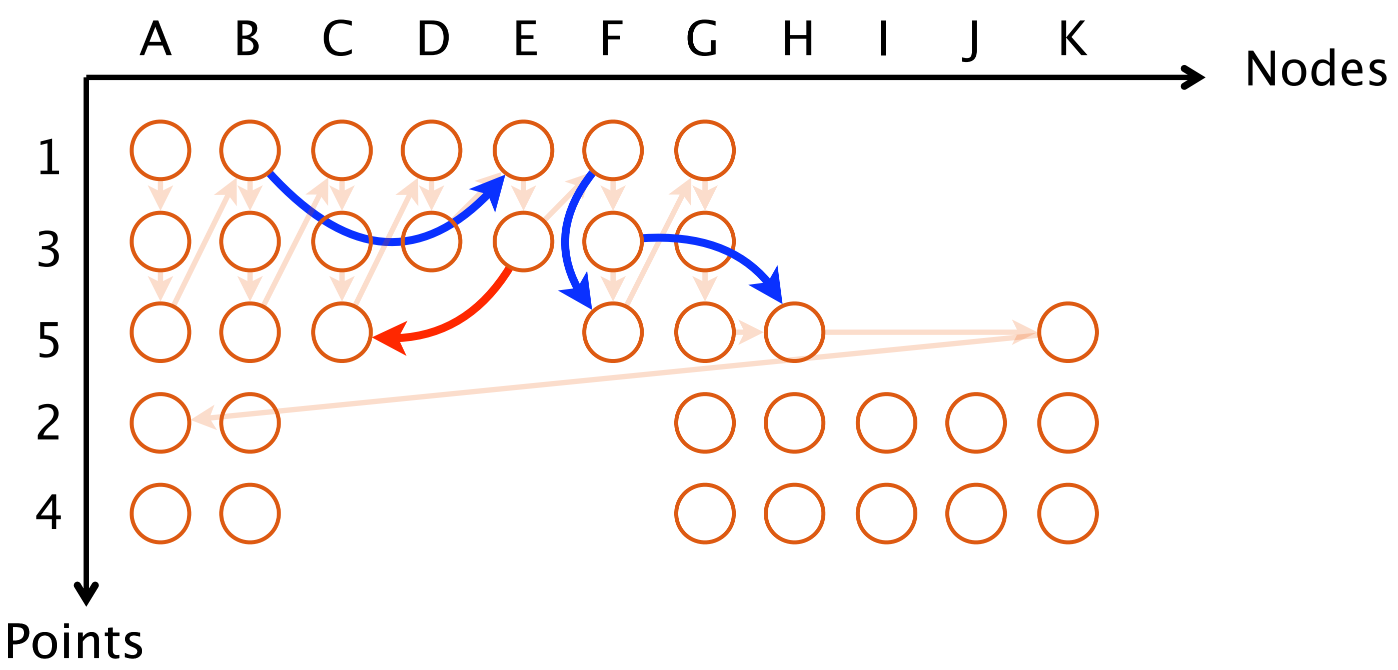
Reasoning about correctness



Dependence preserved by point blocking

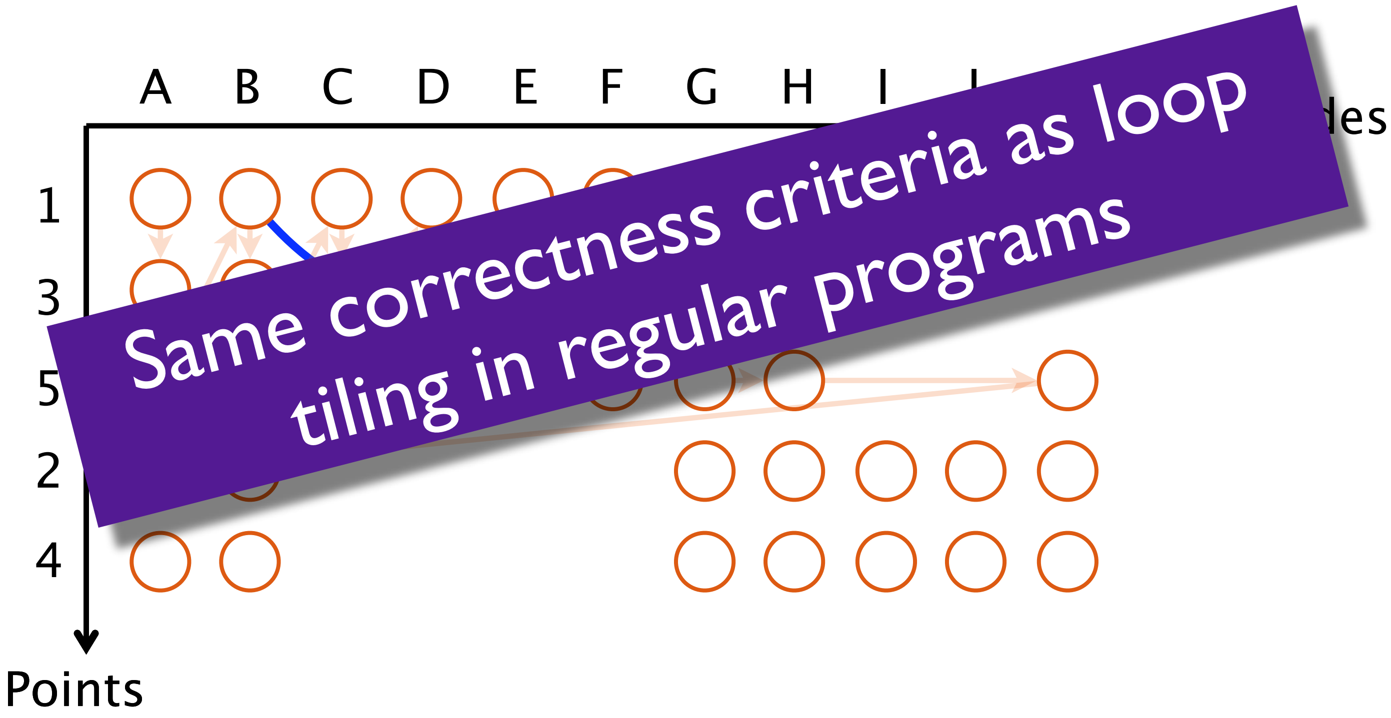
Dependence violated by point blocking

Insight from abstract model



Think about direction vectors: (+, 0), (0, +), (+, +), (+, -)

Insight from abstract model



Think about direction vectors: (+, 0), (0, +), (+, +), (+, -)

Gameplan

- Focus on subset of irregular applications to find common patterns
- Develop models for reasoning about locality
- Design transformations to improve locality
- Determine correctness criteria
- Implement automatic, tuned transformations
- Rinse and repeat

Automation

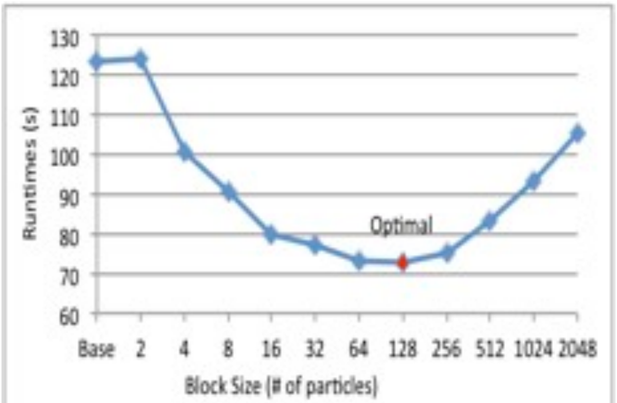
- Look for:
  - Recursive structure
    - Class *c* with field *f* of class *c* (or superclass)
  - Recursive traversal
    - Method *m*, with recursive call `m(c.f, ...)` or `f.m(...)`
  - Enclosing loop (point loop)
- Sufficient condition for correctness: enclosing loop is parallelizable
  - No *inter-point* dependences

Transformation

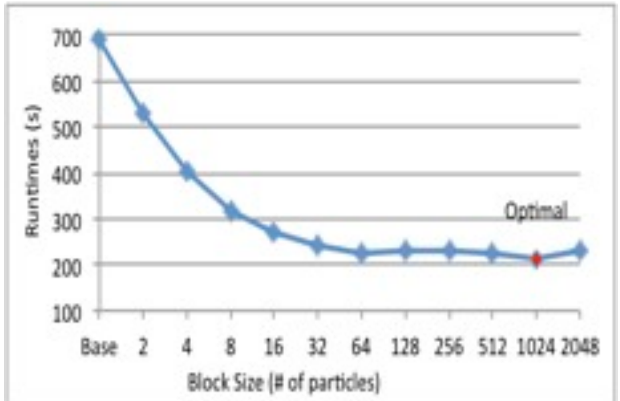
- Engineering tricks
  - Keep track of points that must interact with a given node using block stack
  - Compress block stack at each level
- Tricky details
  - What if (order of) recursion is conditional?
- Parallelization
  - Run multiple point blocks simultaneously

Tuning

- Must choose right block size
  - Too big → doesn't fit in cache
  - Too small → Unnecessary misses in tree



Barnes-Hut



Point Correlation

- Block size is application, architecture and *input* dependent

Tuning

- Instrument code with *run-time* autotuner
  - Hill-climbing approach
  - Random sampling to mitigate input variance
  - Consume no more than 1% of points

Evaluation

- *TreeTiler*
  - Source-to-source transformations in JastAdd
  - Identifies potential loops for point blocking
  - Automatically applies transformation
  - Inserts tuning code
- 5 sample applications: Barnes-Hut, point correlation, nearest neighbor, ray tracing, light cuts

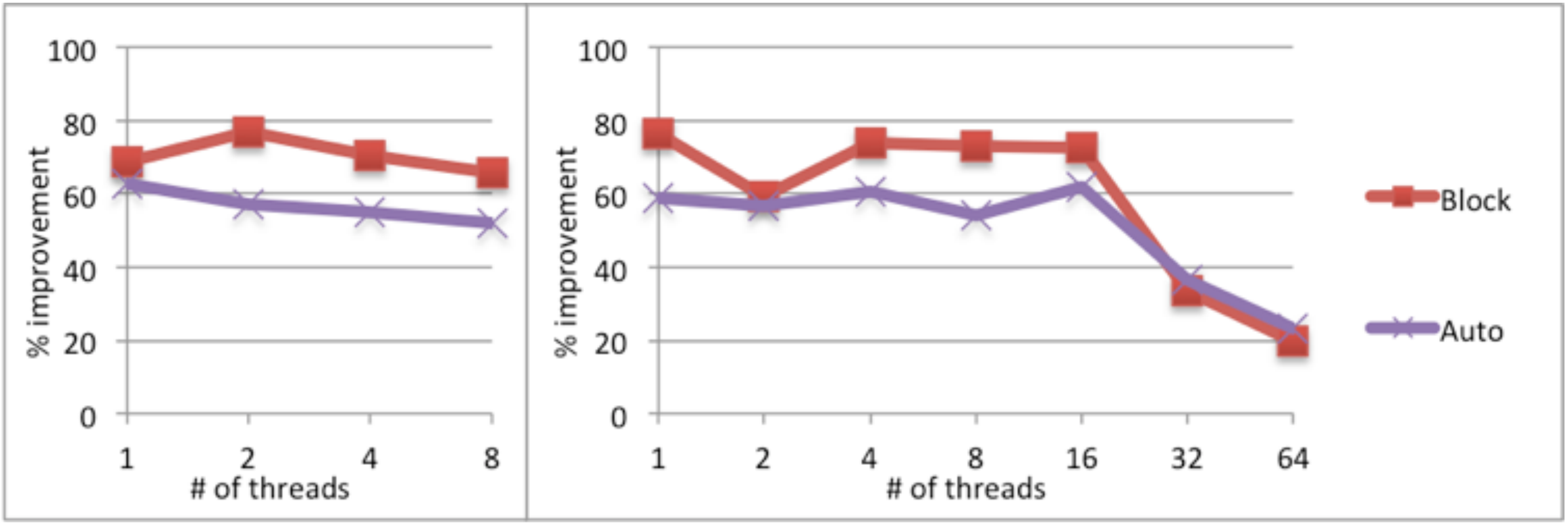
TreeTiler compile time

Application	# Files	Lines of Code	Transform Time (ms)	Total Time (ms)
Barnes-Hut	5	364	8.3	998
Point Correlation	5	390	6.2	975
Nearest Neighbor	3	367	5.4	810
Raytracing	38	3810	10.8	1798
Lightcuts	59	4291	11.2	2342

Methodology

- 3 variants of benchmarks
  - Optimized baseline, "best block", autotuned
- 2 systems
  - Niagara: two 8-core UltraSPARC T2 chips, 8K L1D, 4M shared L2, 1-64 threads
  - Opteron: four dual-core AMD Opteron, 128K L1D, 1M L2, 1-8 threads
- Written in Java and run on Sun HotSpot VM 1.6
- 12GB JVM heap
- Average of latter 7 of 10 runs recorded, GC time excluded

Barnes-Hut

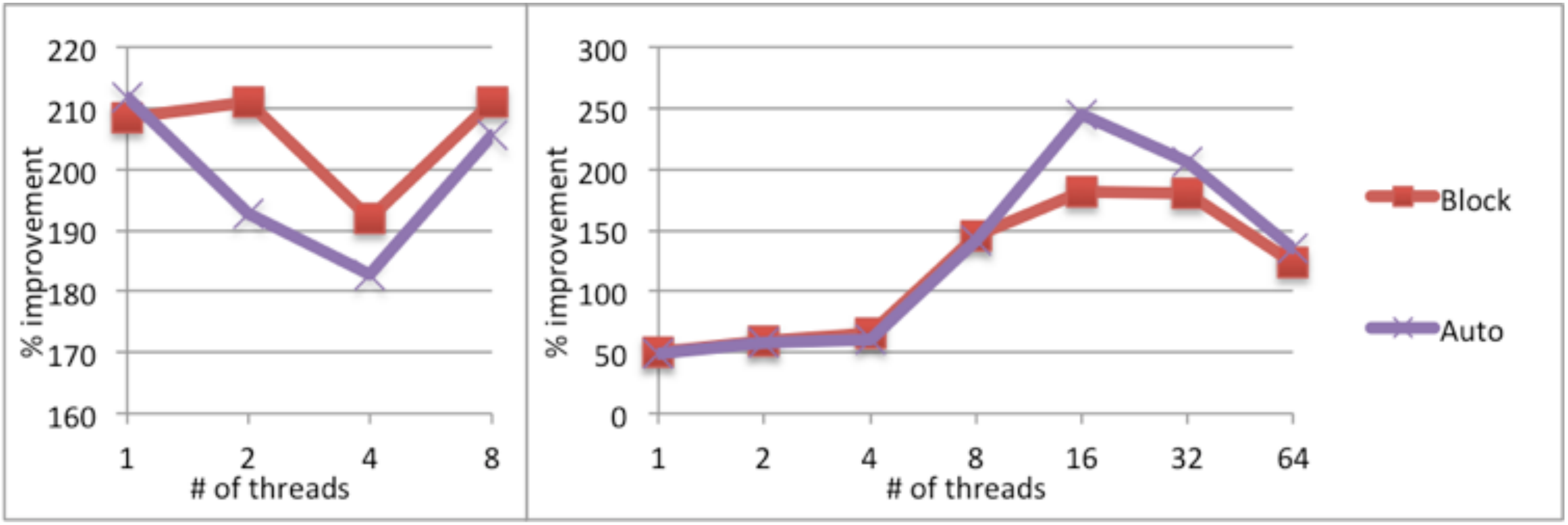


Opteron

Niagara

Input: 1 million bodies

Point correlation



Opteron

Niagara

Input: 1 million points (self-correlation)

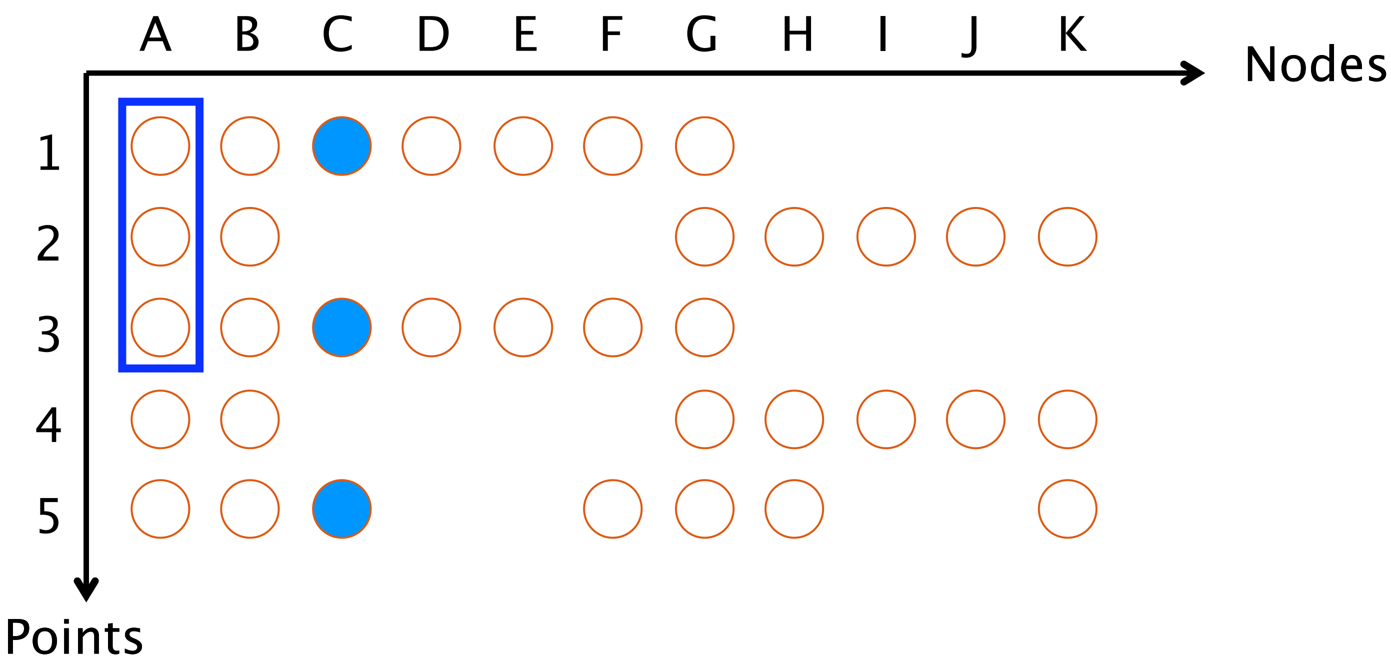
Gameplan

- Focus on subset of irregular applications to find common patterns
- Develop models for reasoning about locality
- Design transformations to improve locality
- Determine correctness criteria
- Implement automatic, tuned transformations
- Rinse and repeat

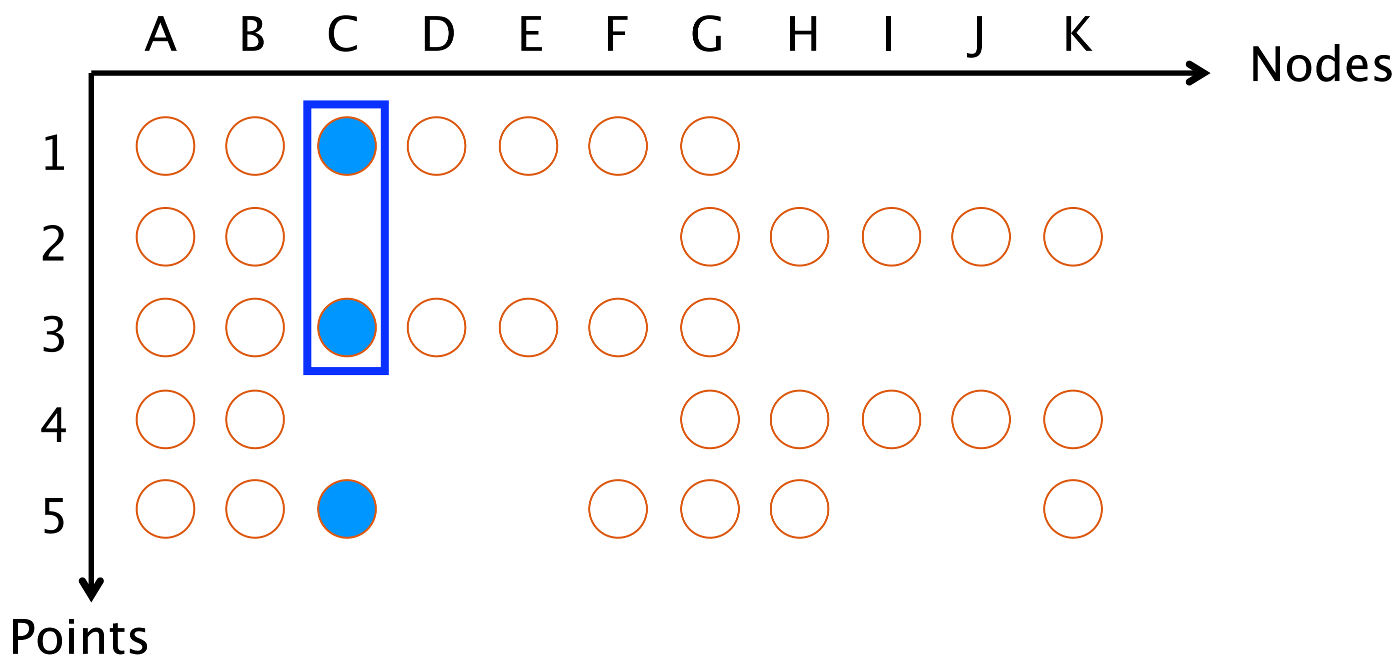
Downsides to point blocking

- Point blocking is very effective on *sorted* inputs
  - Relies on blocks having high occupancy (high *effective block size*)
- But sorting is an application-specific transformation
  - Not very automatic
  - Not always obvious how to sort the points

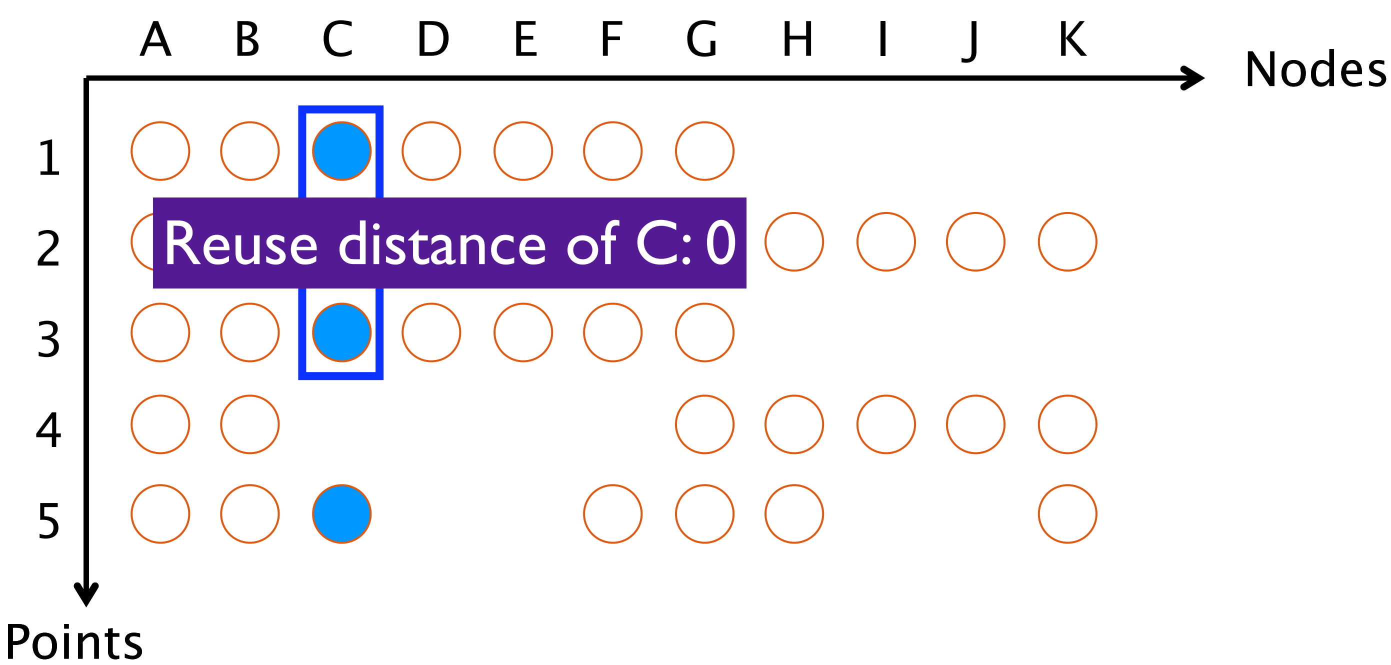
Point blocking without sorting



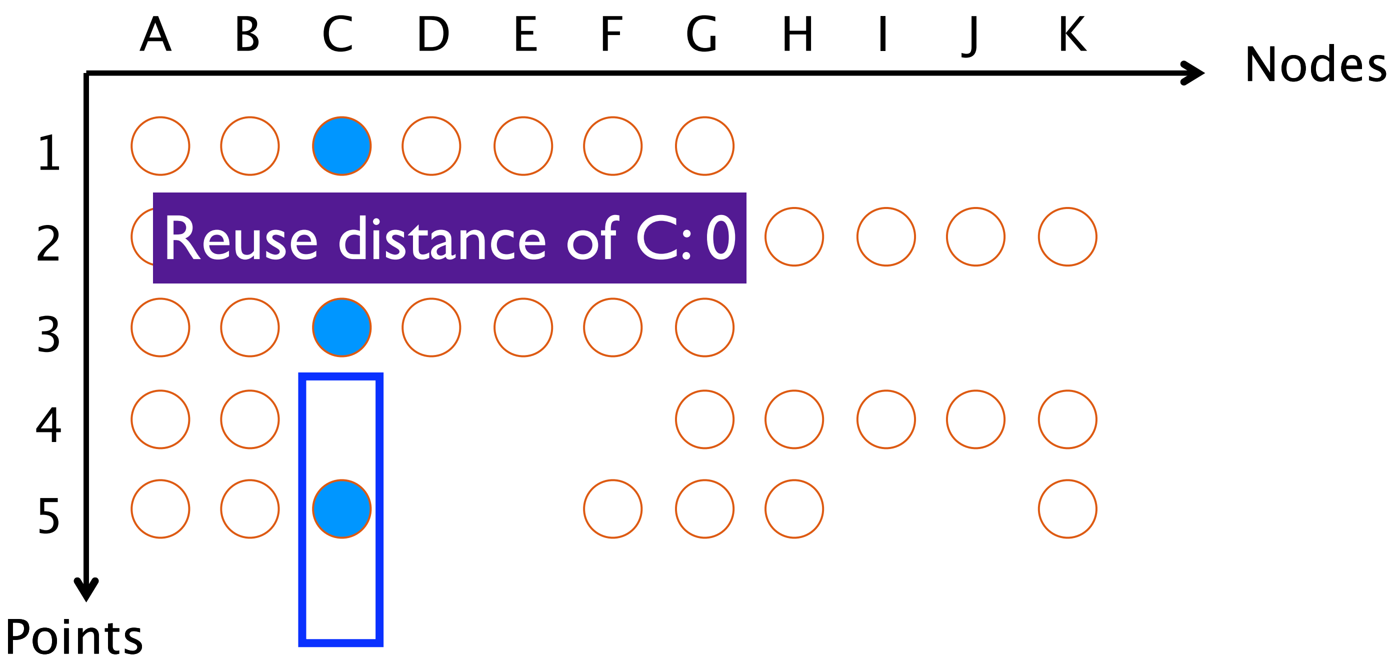
Point blocking without sorting



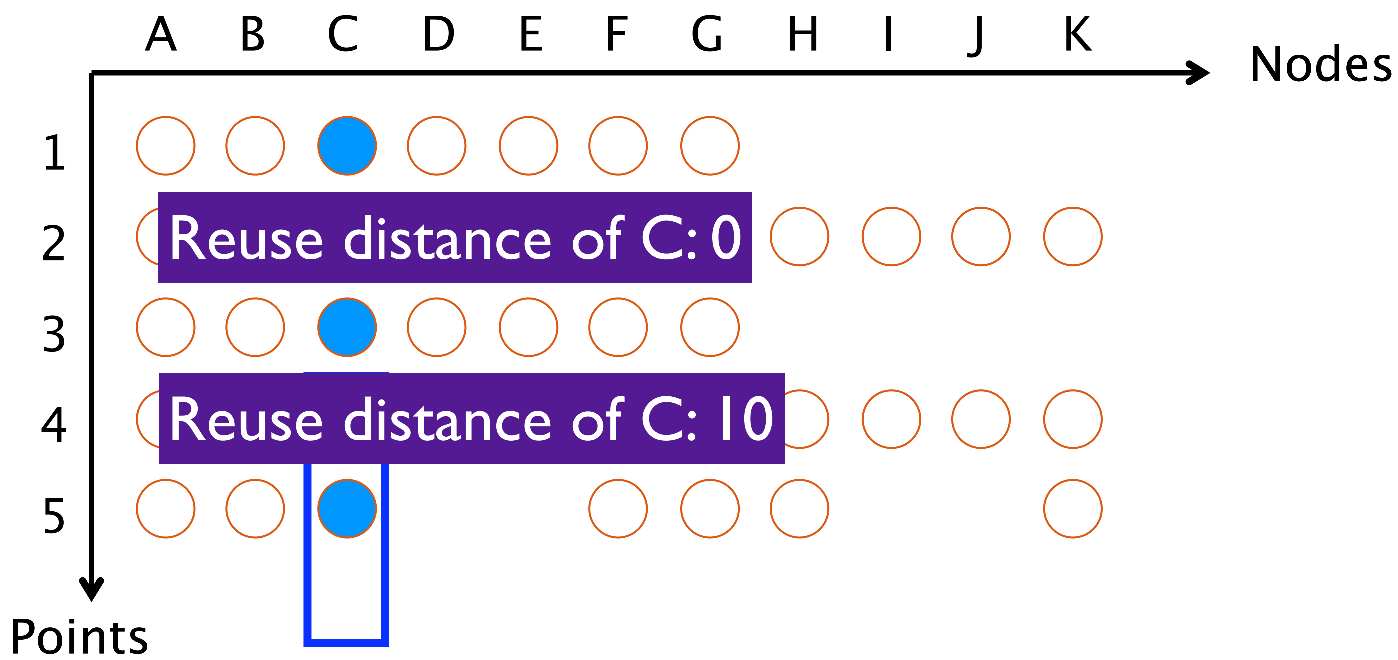
Point blocking without sorting



Point blocking without sorting



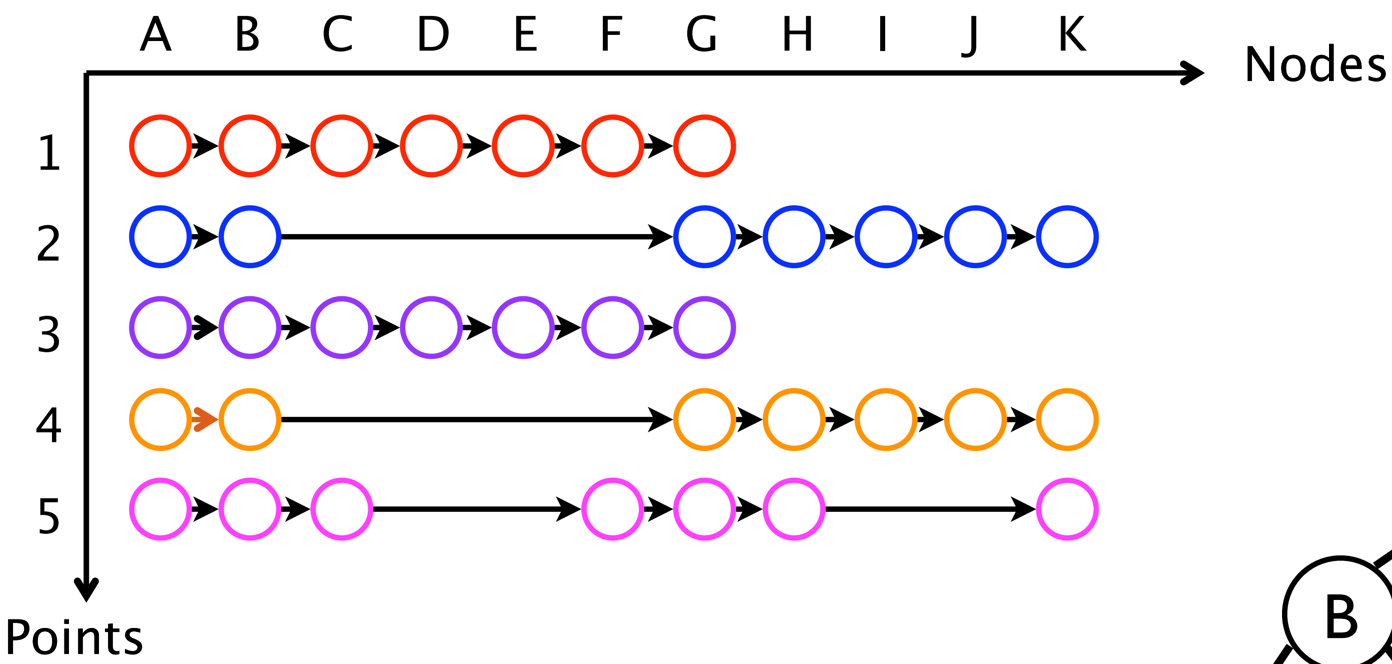
Point blocking without sorting

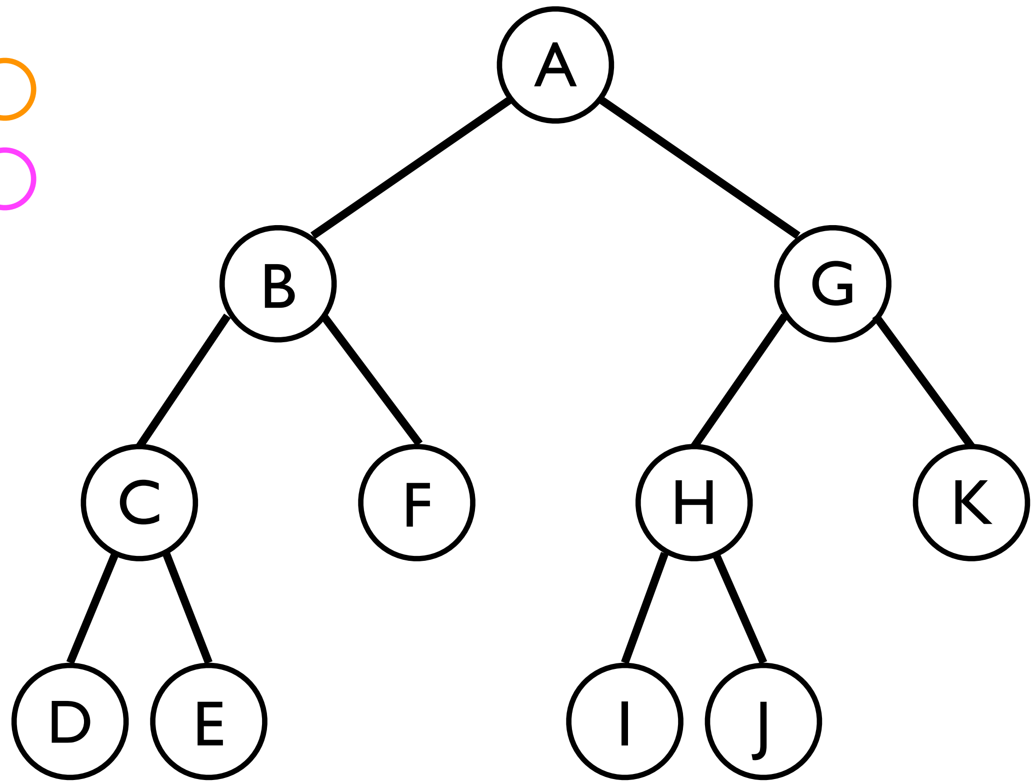


Traversal splicing

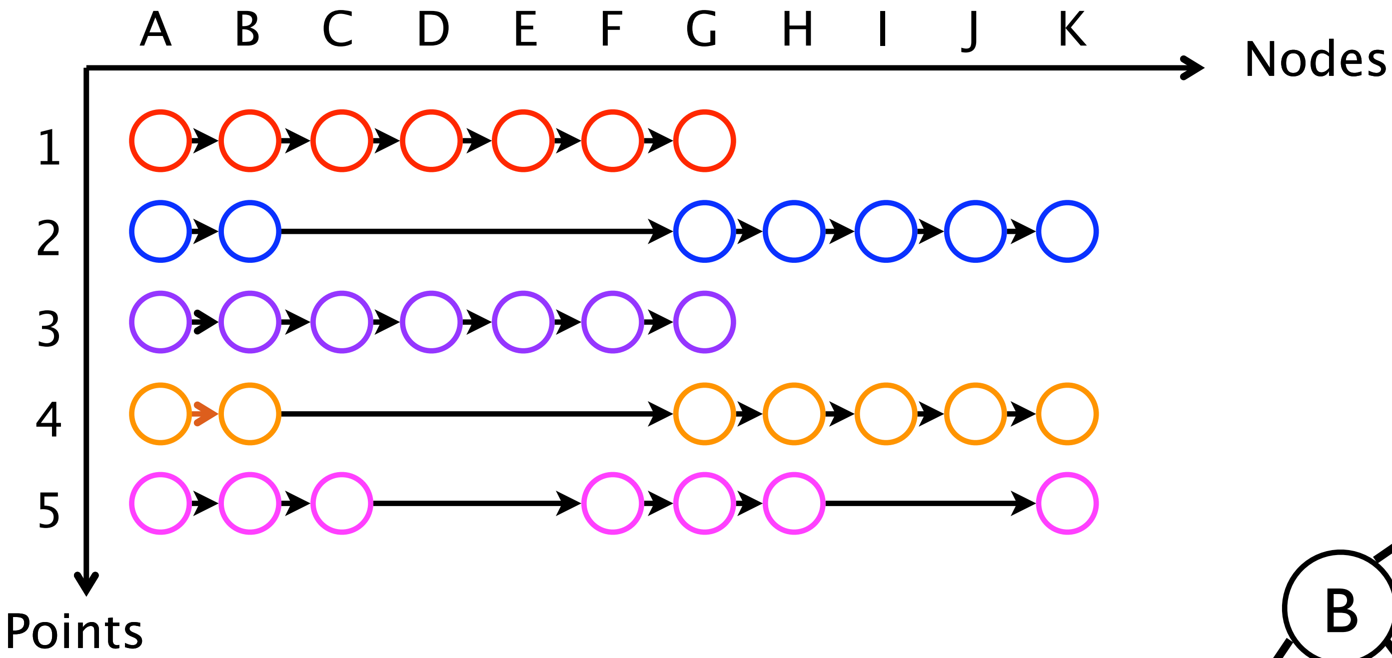
- Tile traversal loop too!
- Then carefully schedule

Traversal splicing

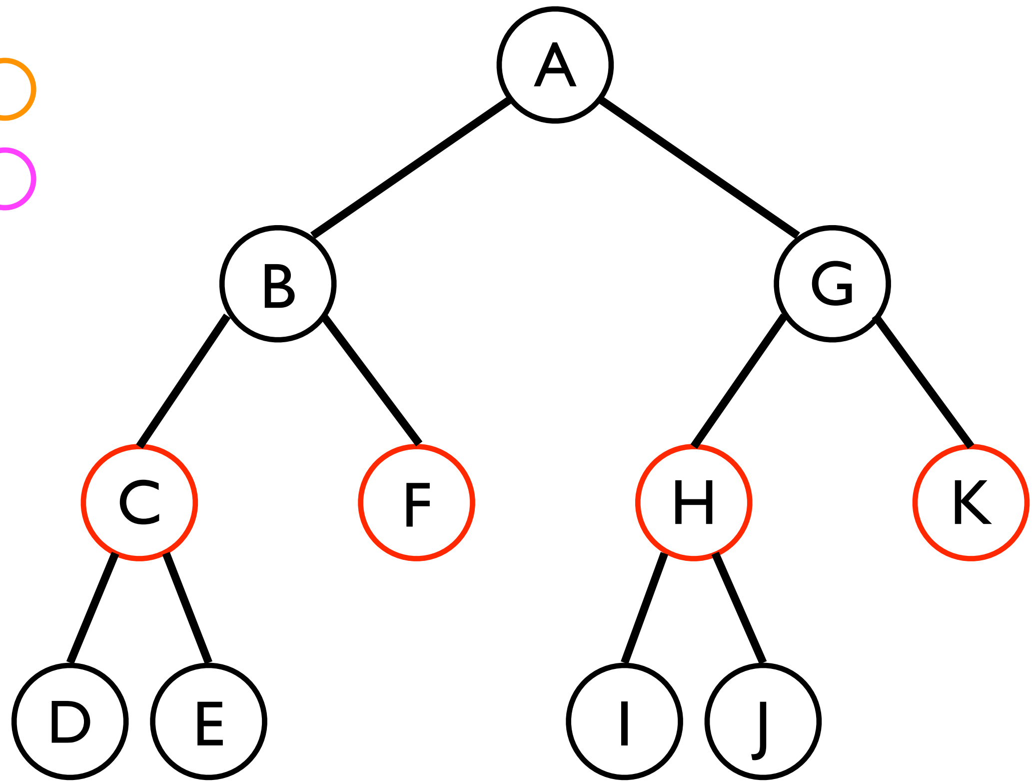




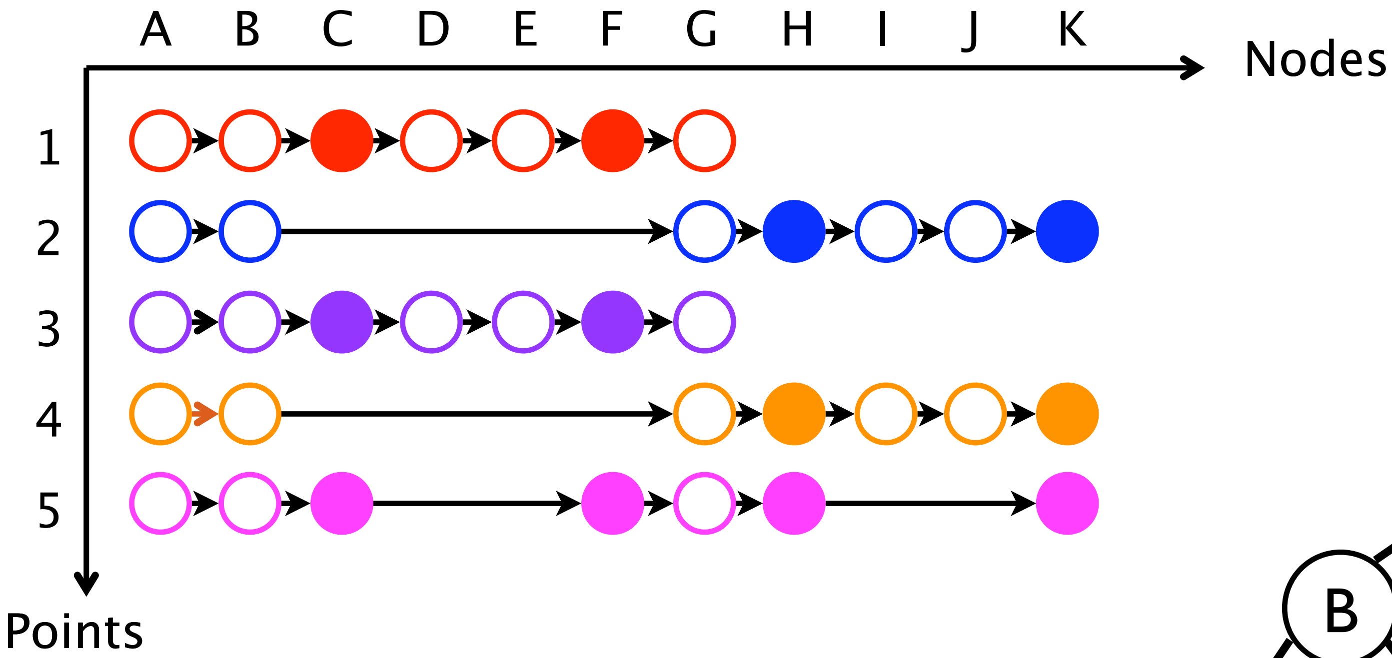
Traversal splicing

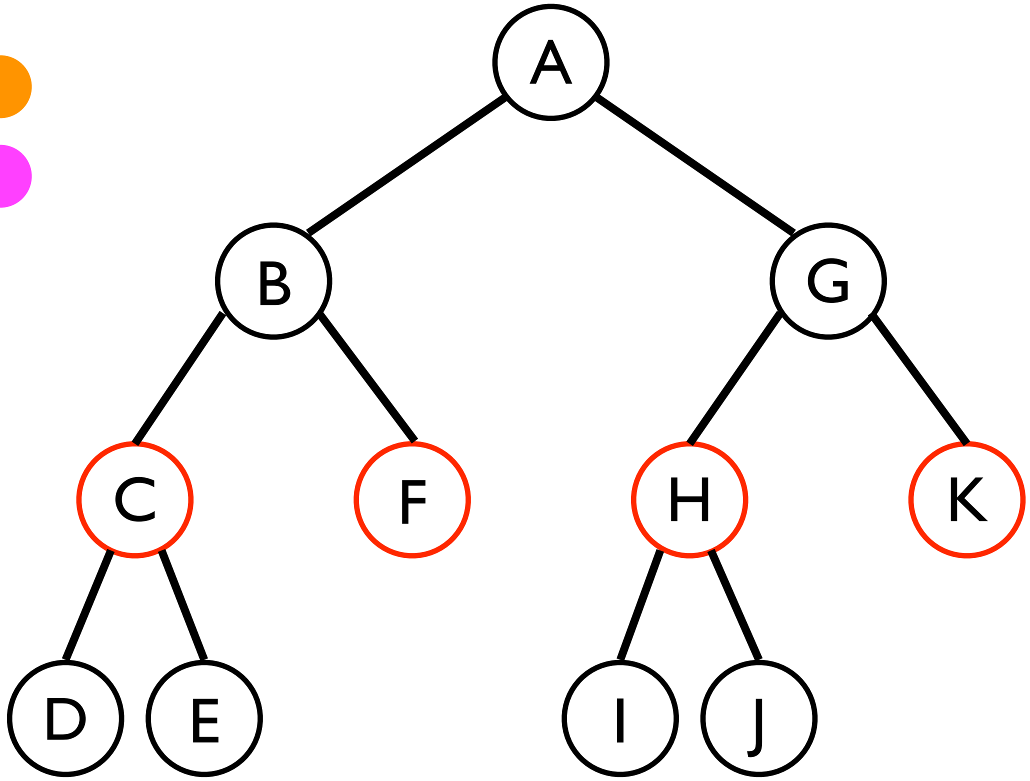


1. Select *splice nodes*



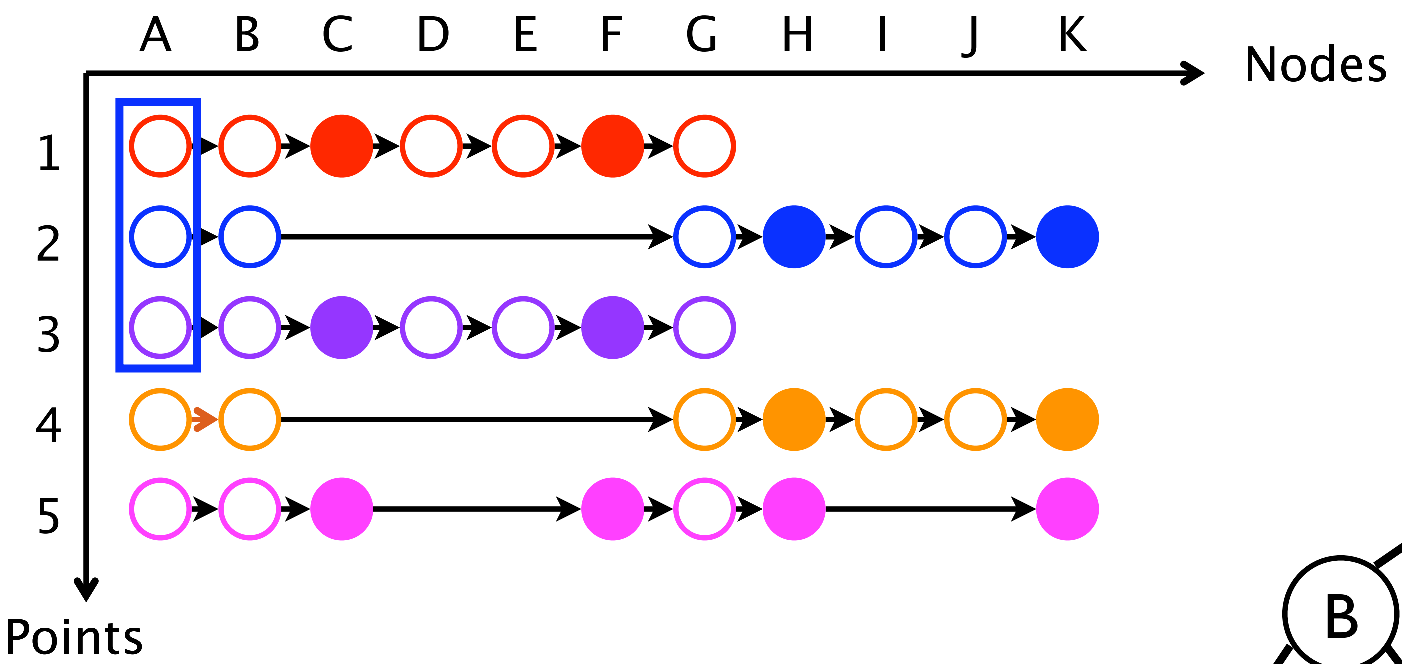
Traversal splicing



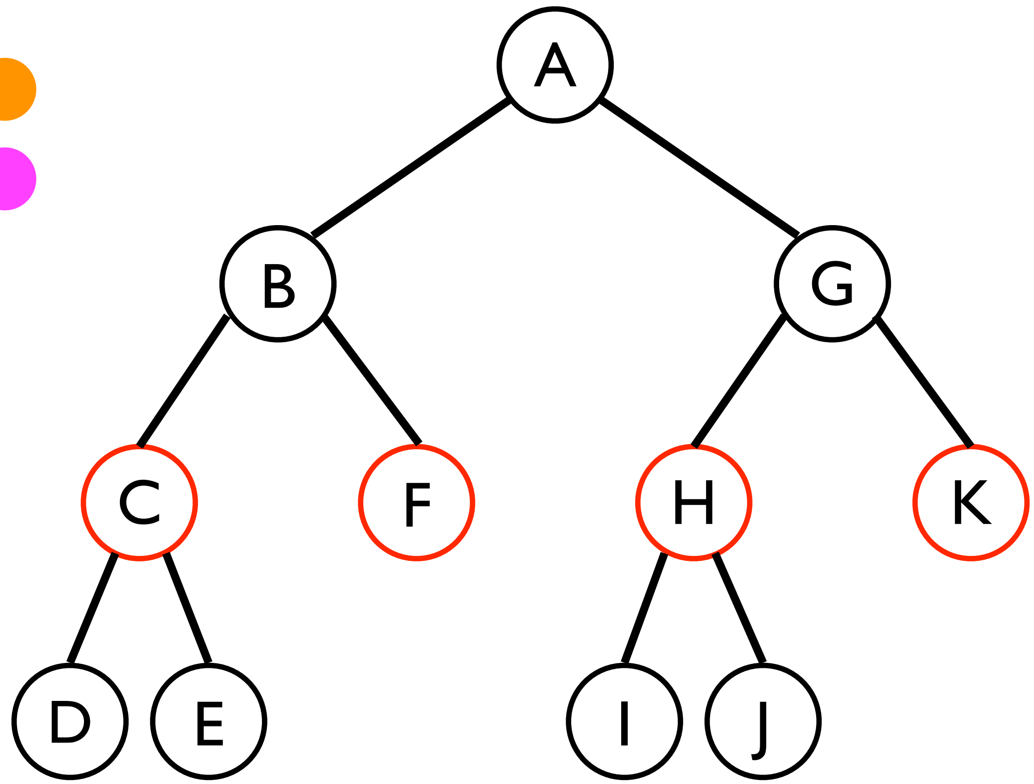


1. Select *splice nodes*
2. "Pause" traversals at splice nodes

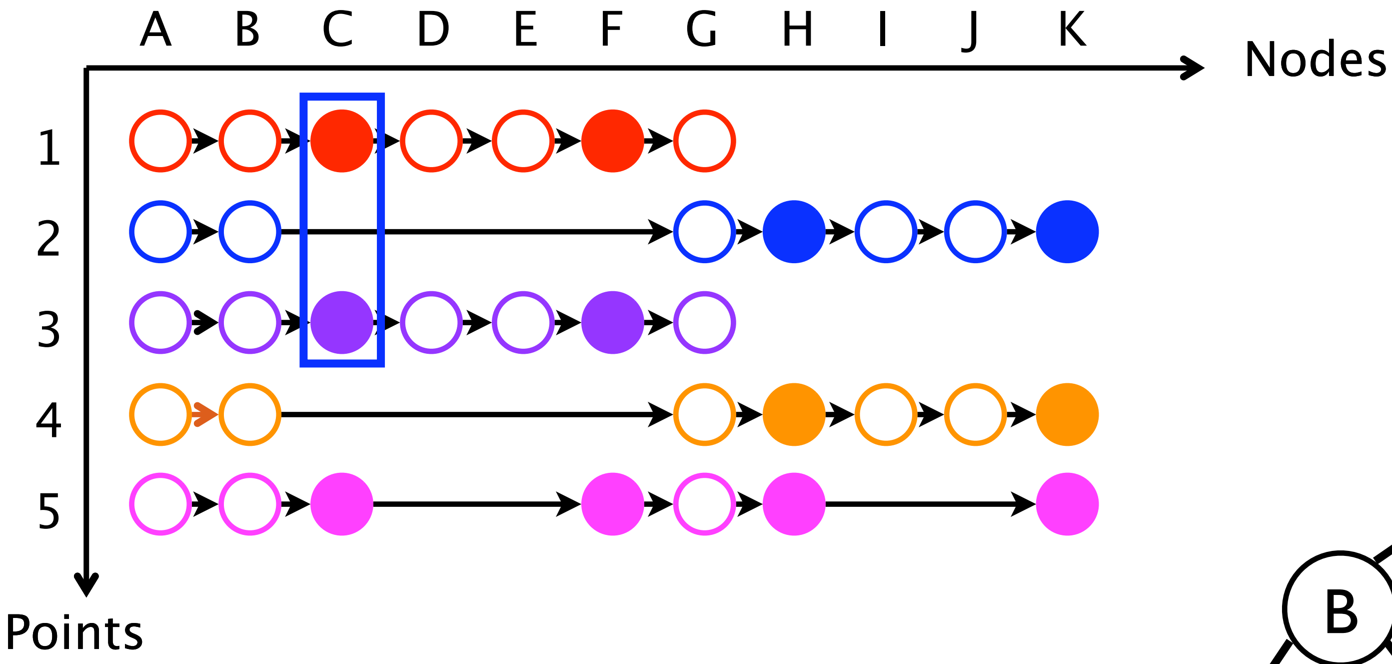
Traversal splicing



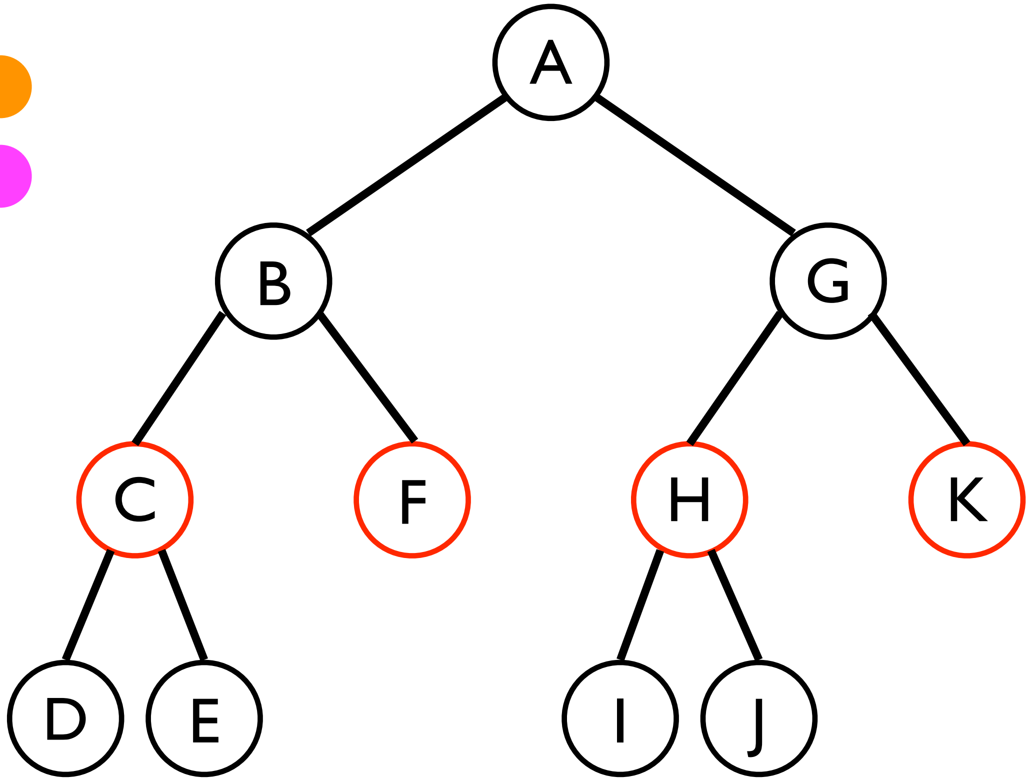
1. Select *splice nodes*
2. “Pause” traversals at splice nodes



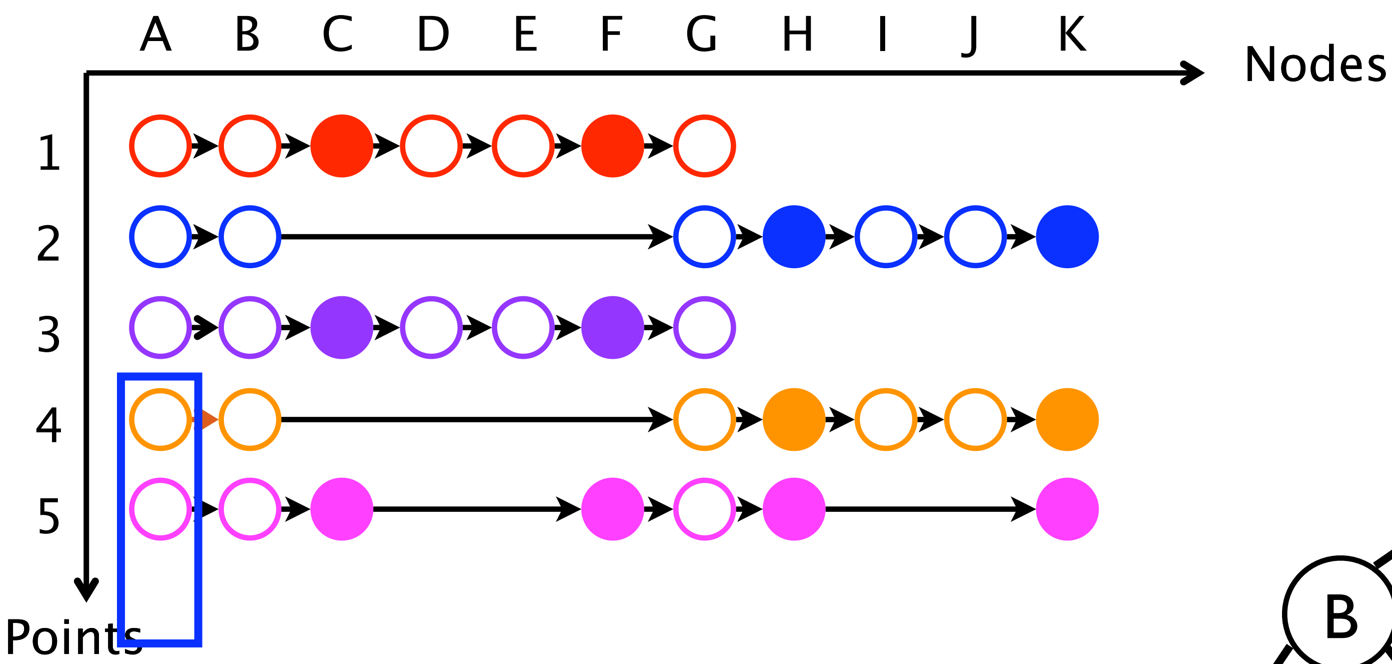
Traversal splicing

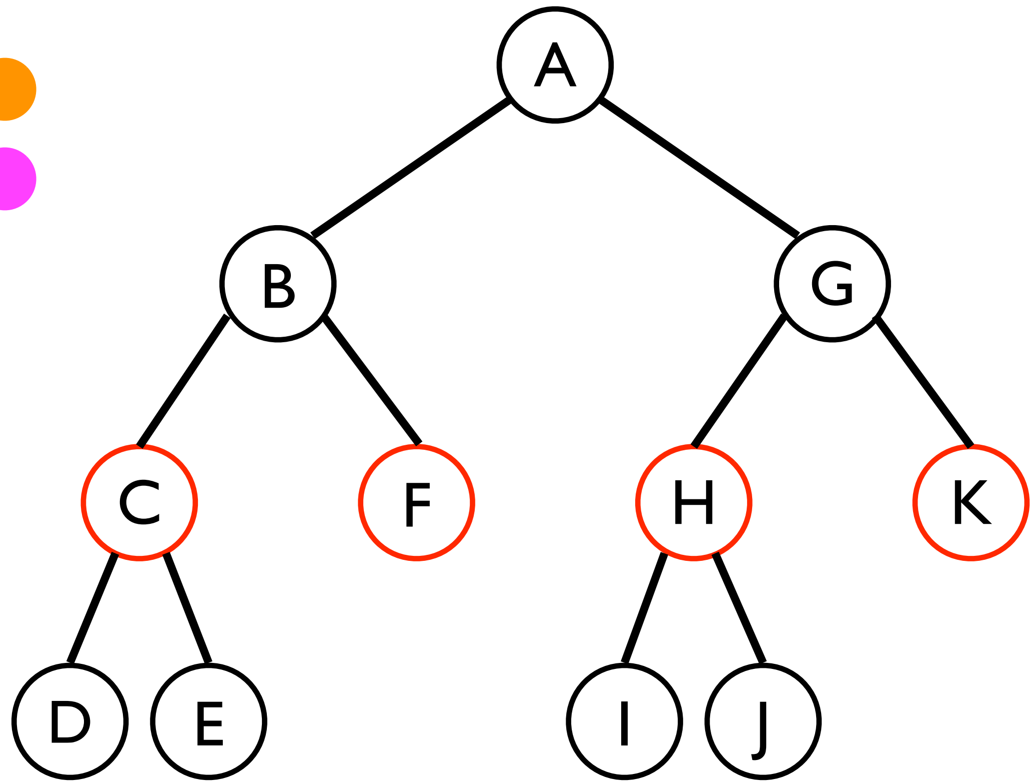


1. Select *splice nodes*
2. "Pause" traversals at splice nodes



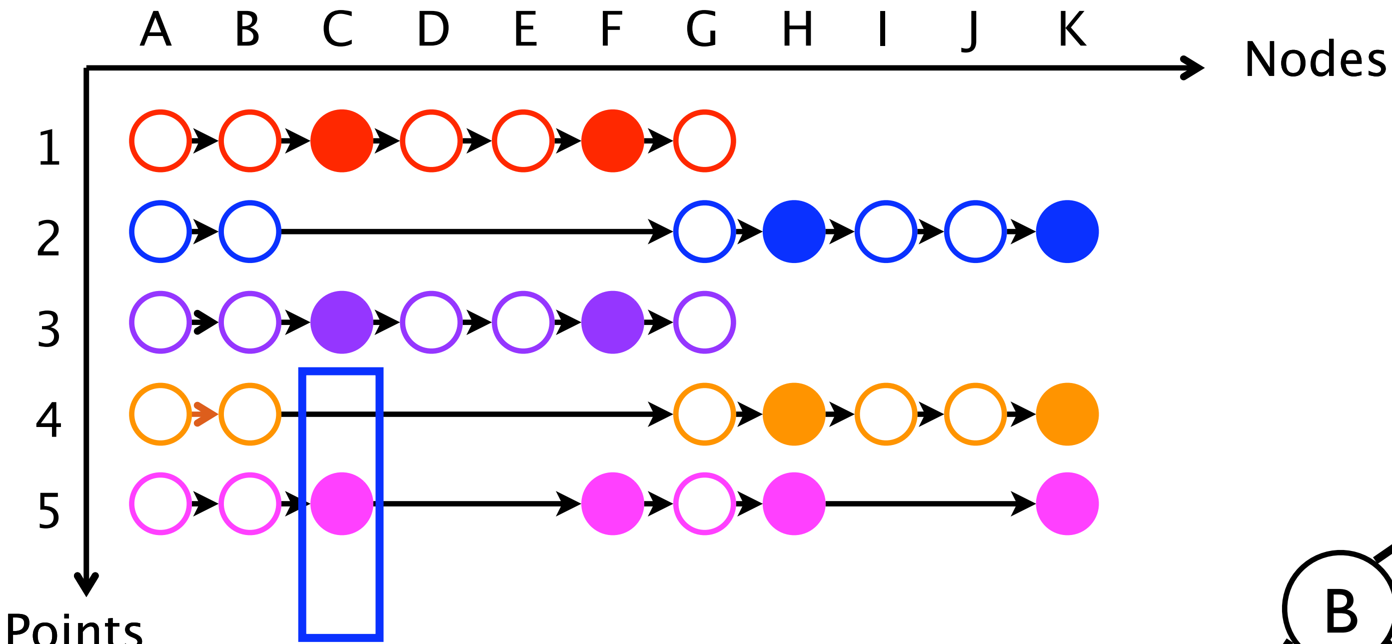
Traversal splicing



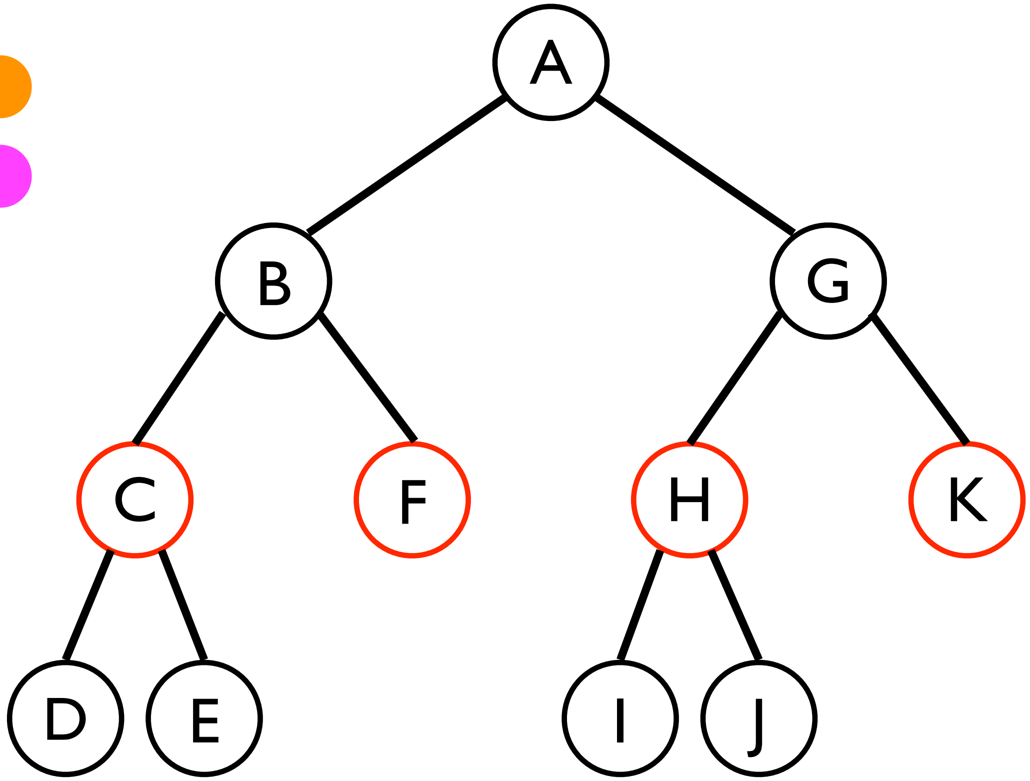


1. Select *splice nodes*
2. "Pause" traversals at splice nodes

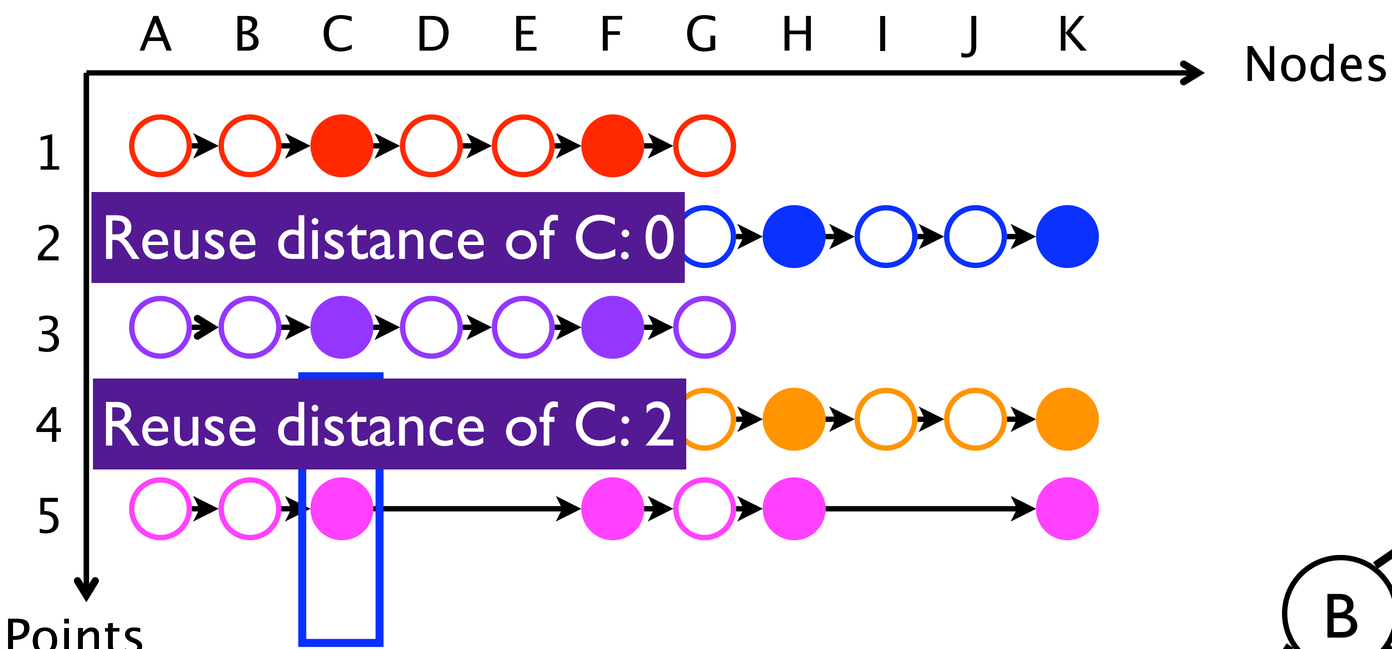
Traversal splicing



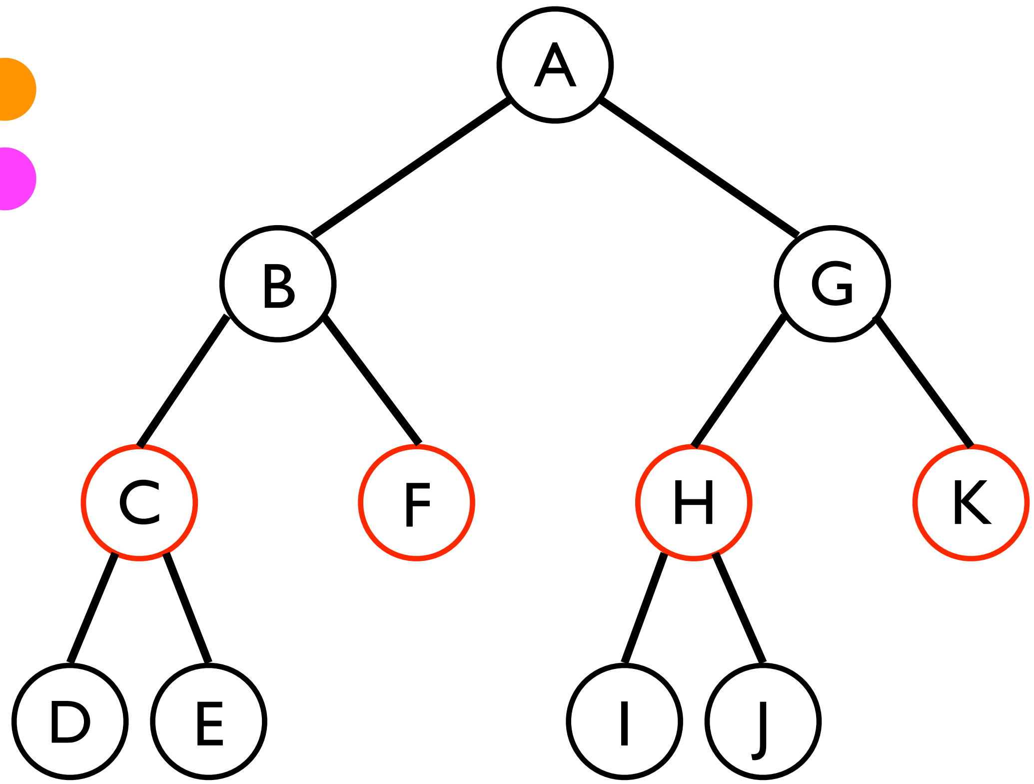
1. Select *splice nodes*
2. "Pause" traversals at splice nodes



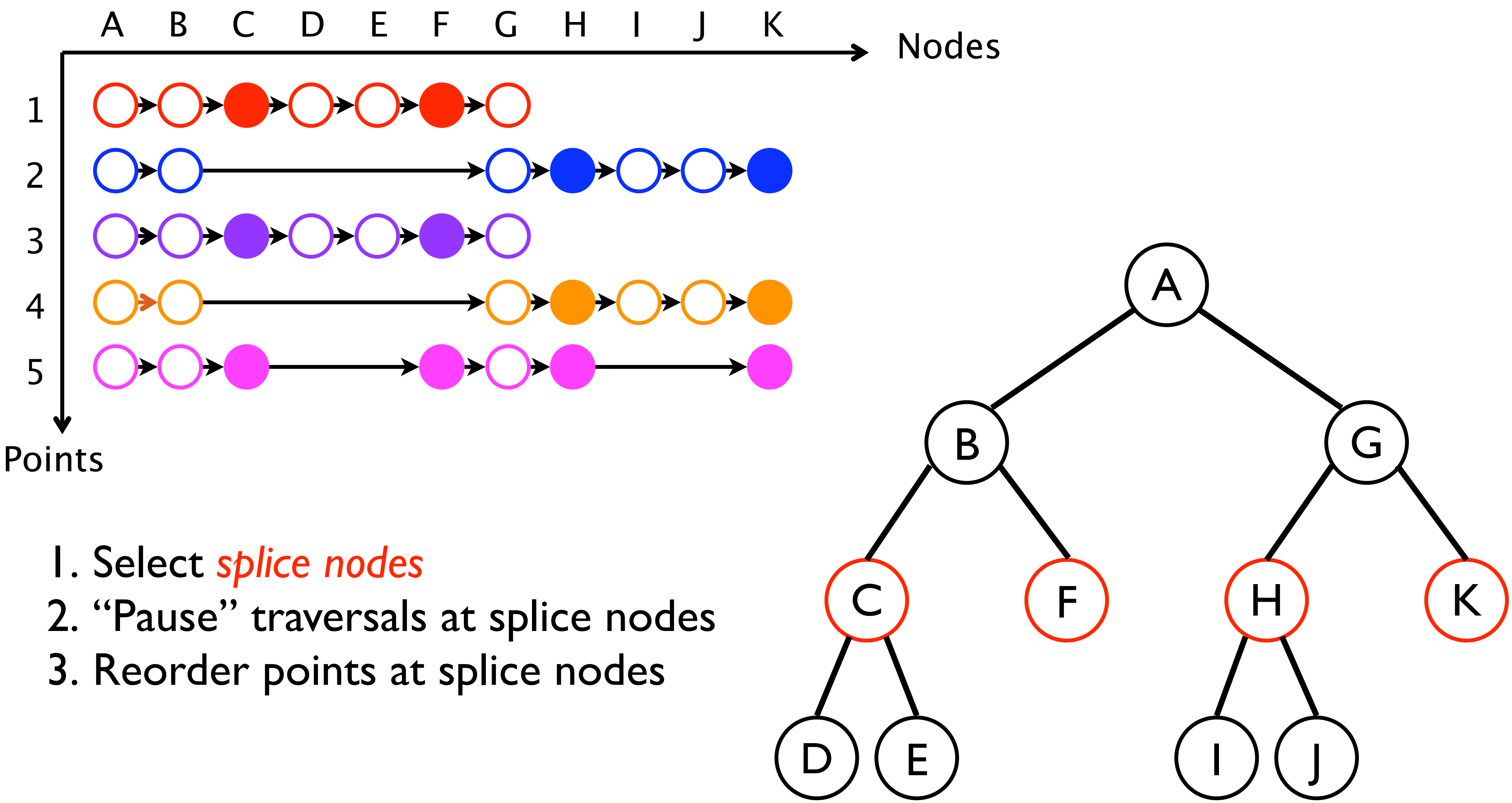
Traversal splicing



1. Select *splice nodes*
2. "Pause" traversals at splice nodes

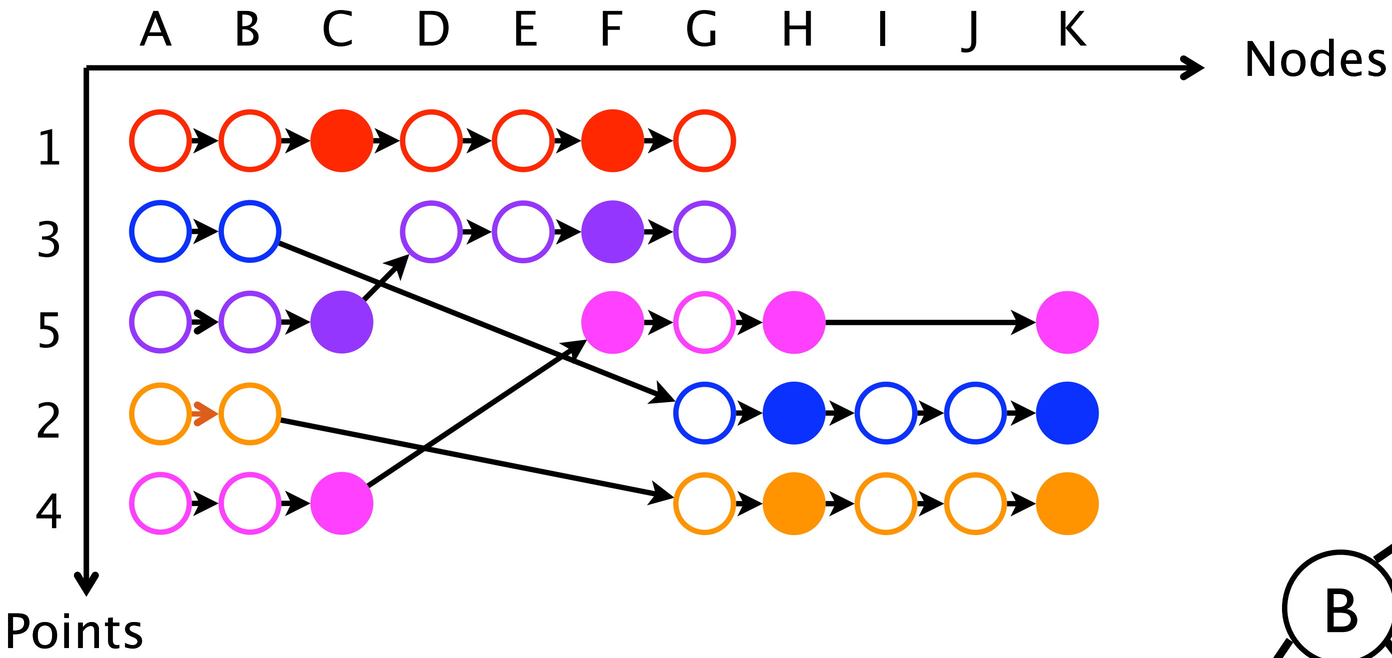


Traversal splicing

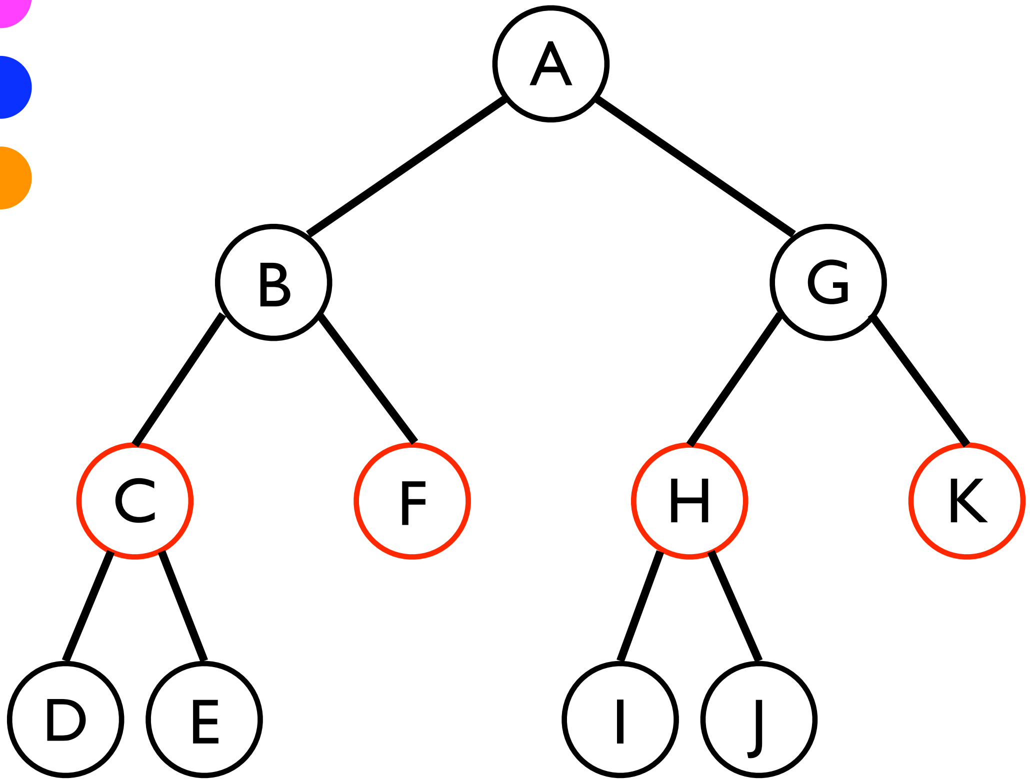


1. Select *splice nodes*
2. “Pause” traversals at splice nodes
3. Reorder points at splice nodes

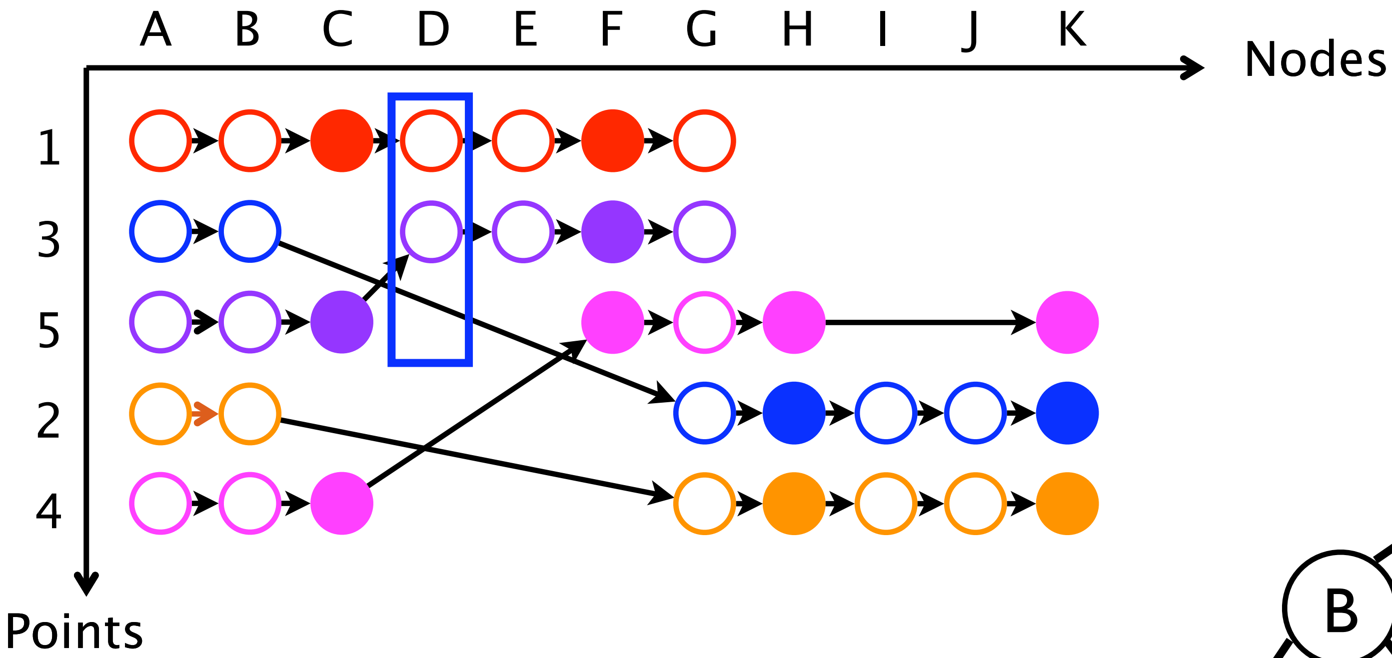
Traversal splicing

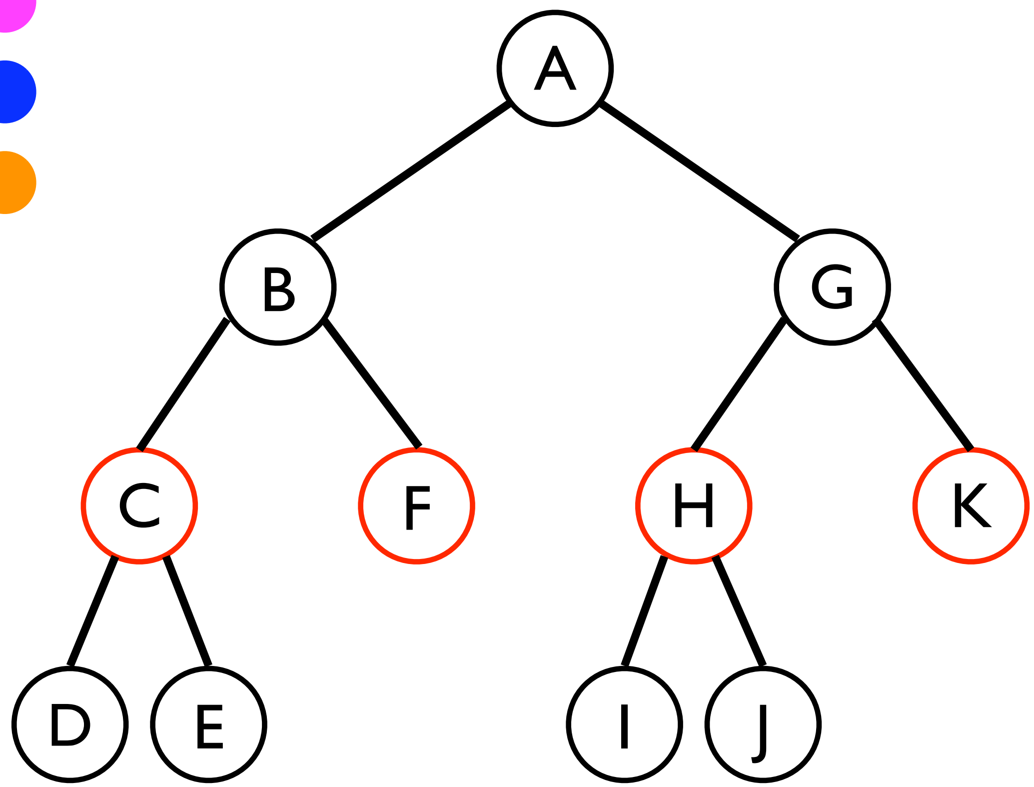


1. Select *splice nodes*
2. “Pause” traversals at splice nodes
3. Reorder points at splice nodes
4. “Restart” traversals



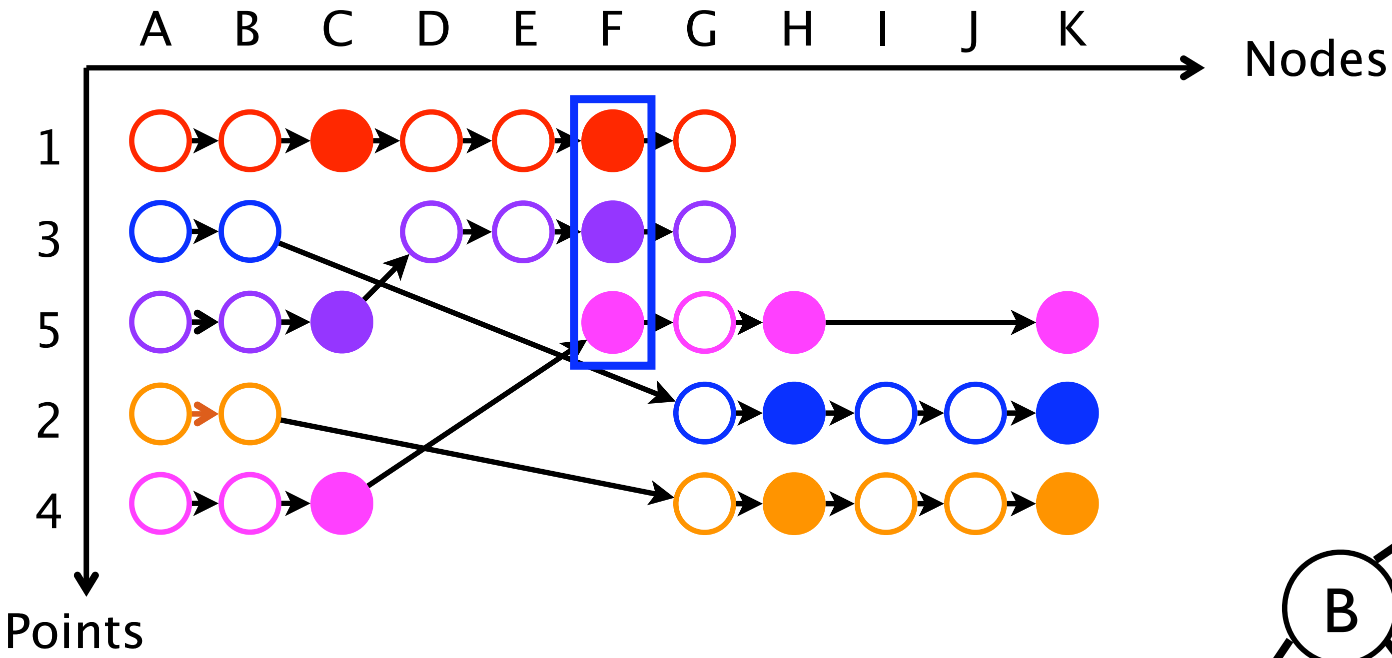
Traversal splicing



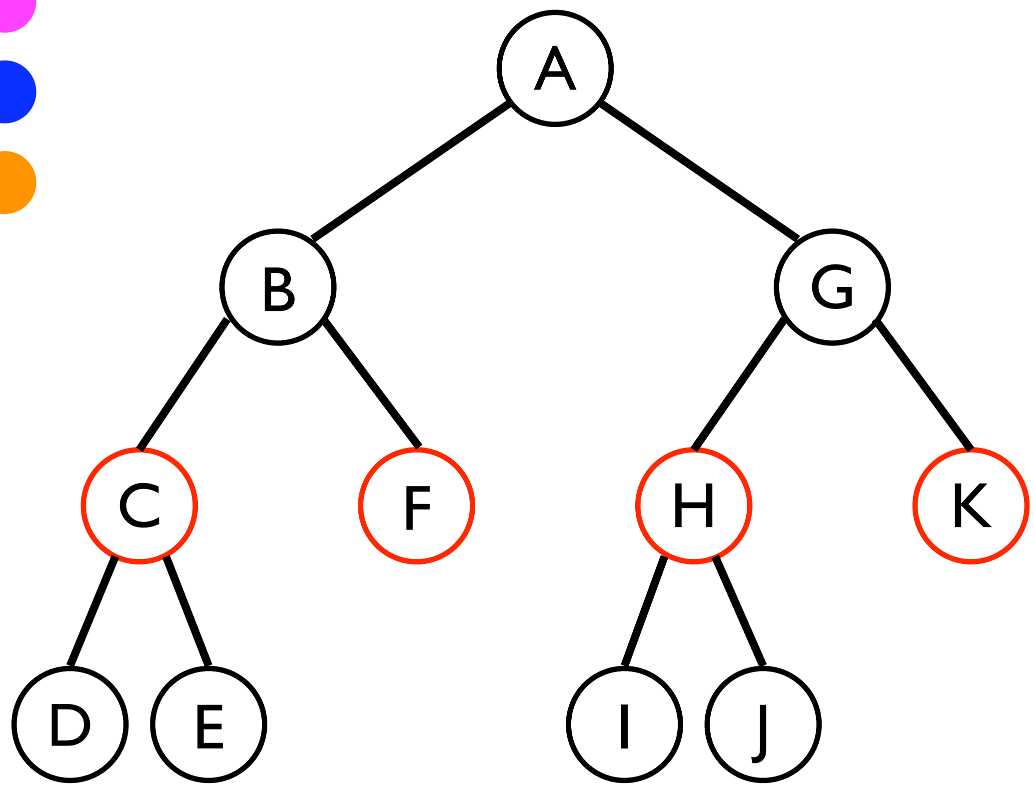


1. Select *splice nodes*
2. “Pause” traversals at splice nodes
3. Reorder points at splice nodes
4. “Restart” traversals

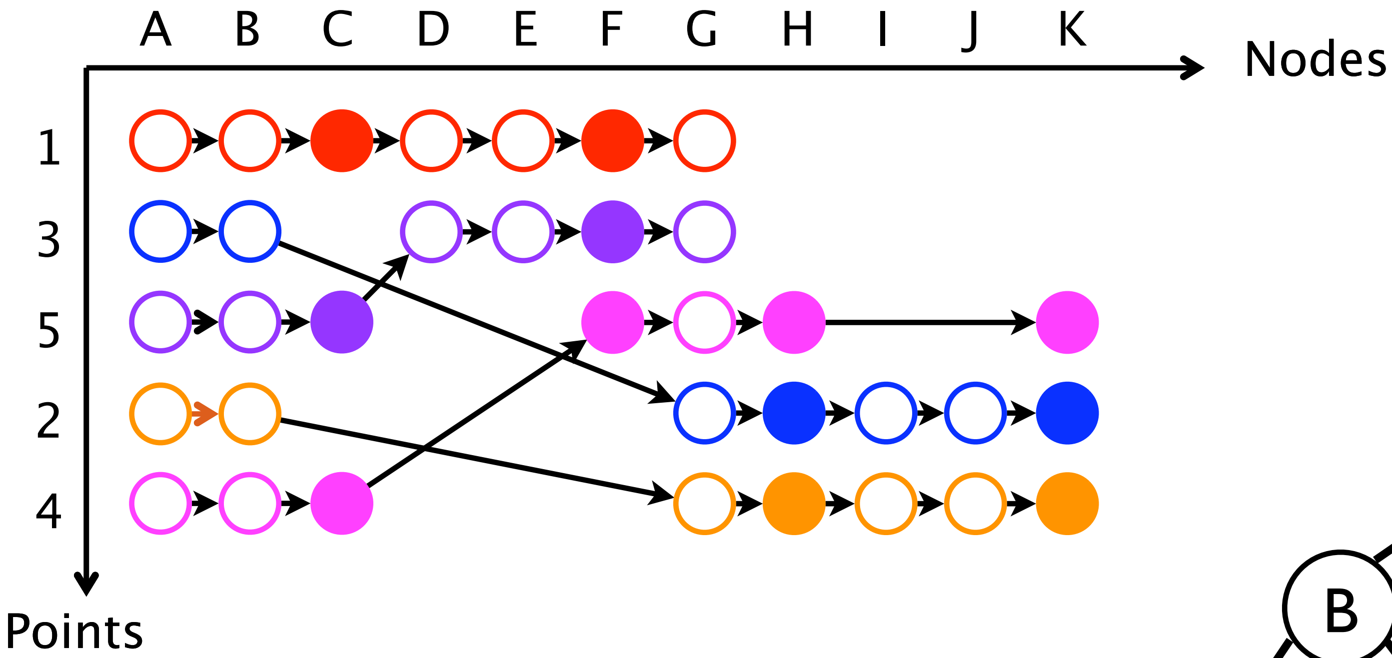
Traversal splicing

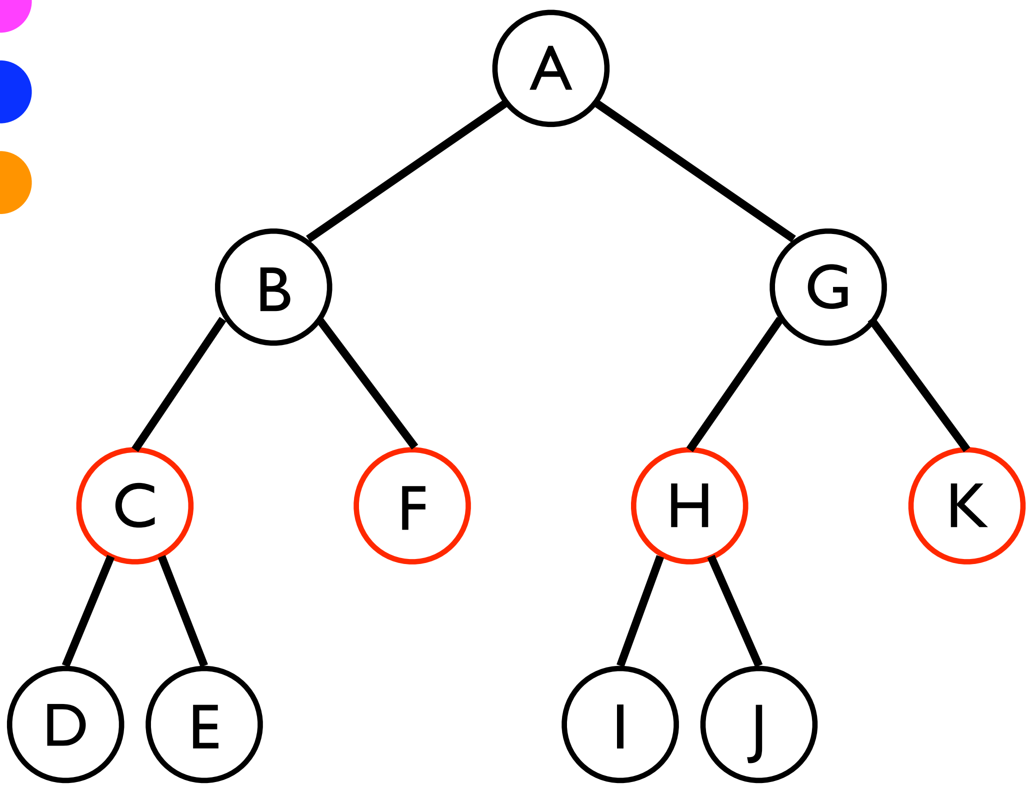


1. Select *splice nodes*
2. “Pause” traversals at splice nodes
3. Reorder points at splice nodes
4. “Restart” traversals



Traversal splicing





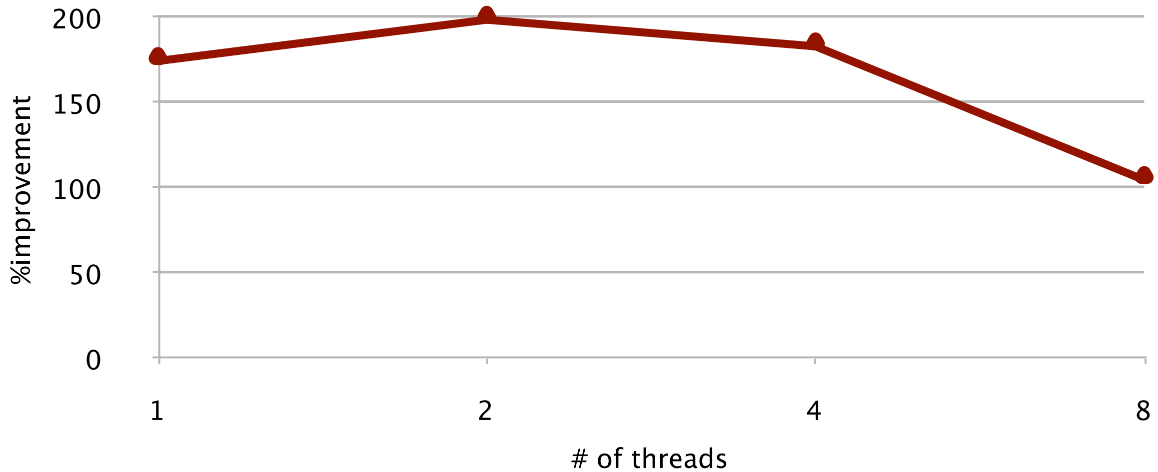
Effects:

- Better locality in tree between splice nodes
- On-demand reordering improves effective block size

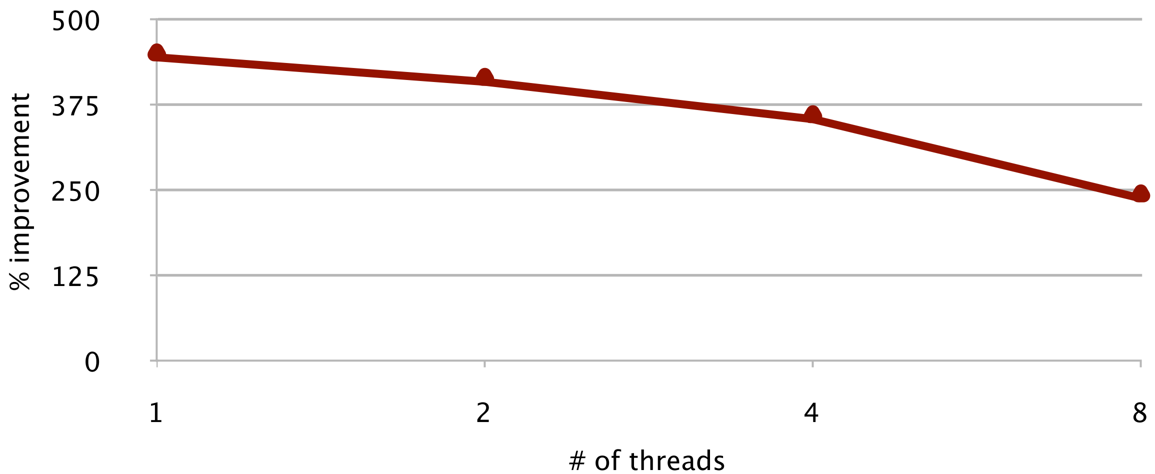
Correctness issues

- Each point traverses tree in same order
  - Intra-point dependences satisfied
- Order that points visit a particular node is changed
  - Inter-point dependences may not be satisfied
- Necessary and sufficient condition: parallelizability

Results



Nearest neighbor
1 million points (unsorted)



Point correlation
1 million points (unsorted)

Current and future work

- Automate and tune traversal splicing
- More sophisticated correctness analyses
  - Necessary and sufficient conditions
- More general algorithms
- Tuning models
- Different platforms
  - Some early success in using techniques to map applications to GPUs

Conclusions

- Irregular algorithms are a fertile ground for locality optimizations
- Need to consider applications at the right level of abstraction
  - Informs transformations, correctness criteria, locality effects
- Can automatically apply locality-enhancing transformations to irregular algorithms and achieve significant performance improvements

Acknowledgments

- Youngjoon Jo
- Michael Goldfarb