11-0: **First-Pass Collision**

- We now know how to to pairwise collision
  - Fast, less accurate AABB collision
  - Slower separating axis with arbitrary bounding shapes
- Which elements do we compare?

11-1: **First-Pass Collision**

- Which elements do we compare?
  - Brute force

```
foreach (WorldObject o in mElements)
   foreach (WorldObject other in mElements)
      if (collide(o, other))
         handle collision
```

  - Error!

11-2: **First-Pass Collision**

- Which elements do we compare?
  - Brute force

```
foreach (WorldObject o in mElements)
   foreach (WorldObject other in mElements)
      if (other != o && collide(o, other))
         handle collision
```

  - Correct, can make it slightly more efficient ...

11-3: **First-Pass Collision**

- Which elements do we compare?
  - Brute force

```
for (i = 0; i < mElements.Count; i++)
   for (k = i+1; k < mElements.Count; k++)
      if collide(mElements[i], mElements[k])
         handle collision
```

  - Running time still $\Theta(n^2)$, not practical for large n.

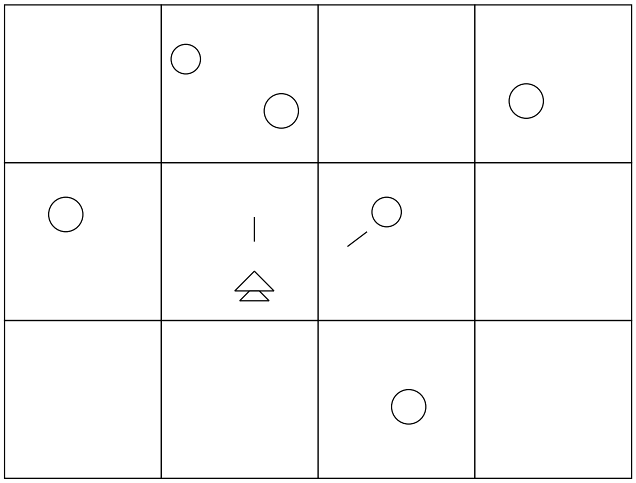
11-4: **First-Pass Collision**

- How can we do better?

11-5: **Grid**

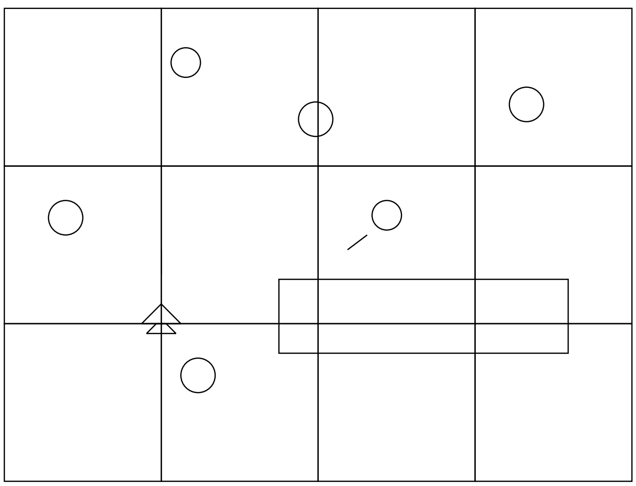
- Separate world into a grid
  - Each grid element stores a list of all elements at that grid location
- To do collision, you only need to look at the elements in the same grid location

11-6: **Grid**



Problems with this approach?

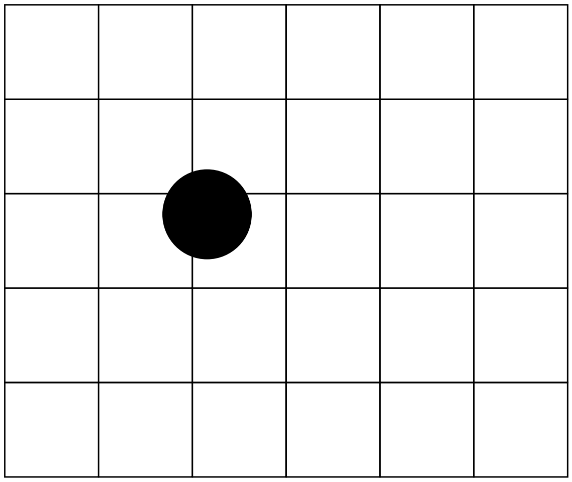
11-7: **Grid**



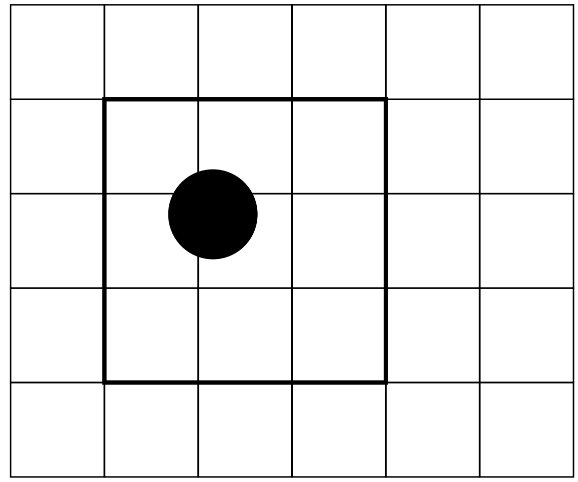
Elements can overlap grid entries.

11-8: **Grid**

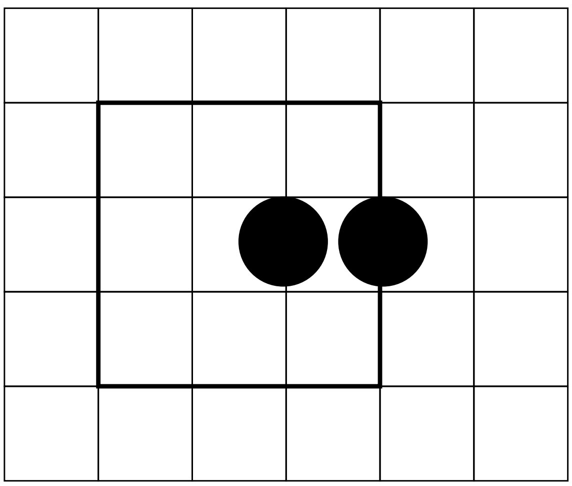
- If:
  - Each world element is smaller than a grid square
  - Each world element is stored in the grid square that contains the center of the object
- Given an object o in our world, how do we determine which elements to we need to check against o for intersection / collision?



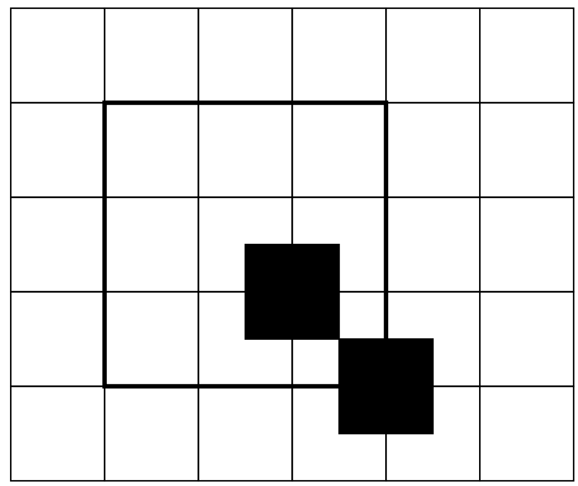
11-9: **Grid**



11-10: **Grid**



11-11: **Grid**



11-12: **Grid**

11-13: **Grid**

- Object o is stored at grid location $[x, y]$
  - Need to consider 9 grid locations for intersection
    - $[x-1, y-1]$, $[x, y-1]$, $[x+1, y-1]$
    - $[x-1, y]$, $[x, y]$, $[x+1, y]$
    - $[x-1, y+1]$, $[x, y+1]$, $[x+1, y+1]$

11-14: **Grid**

- Implementation Details
  - Pick a grid size, make all grid elements the same size
  - 2D array of lists, each list stores elements in that grid element
  - Finding the grid location of an object is easy (how would you do it?)
  - As objects move around in the world, may need to change grid locations

11-15: **Grid**

- Implementation Details

- Don't want to be allocating / deallocating all of the time
- Lists should be lightweight (arrays are likely a good idea) – If you have enough memory, each cell location can have a fixed array size. (May need to deal with overflow – either per-grid cell, or as a global overflow list)
- If arrays are unordered, adding and removing elements is fast (how?)

11-16: **Grid**

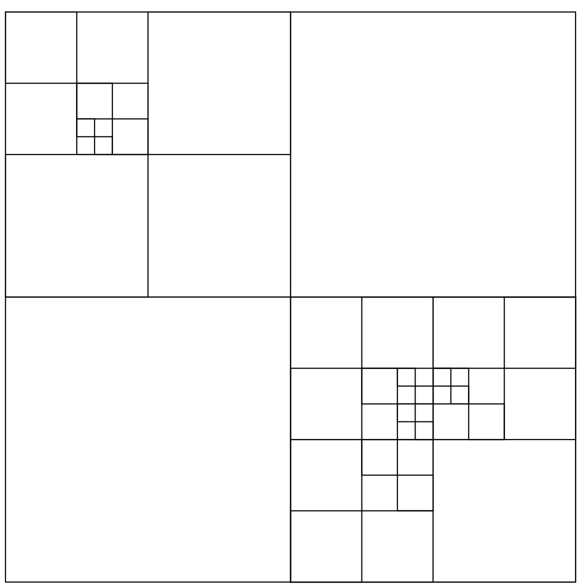
- Implementation Details
  - Don't want to be allocating / deallocating all of the time if you don't need to
  - Moving an element from one grid cell to another can be done quickly, assuming each grid cell doesn't hold too many elements (how would you do it?)

11-17: **Grid**

- Problems with this method?

11-18: **Grid**

- Problems with this method?
  - Large objects require large grid sizes, may need to check large number of elements to do collision
    - If large elements are static (big platforms, etc) we can special case them – split them into smaller objects, place them in several grid locations, etc
  - Big, sparse worlds require large (mostly empty) grids
- Solution: Use non-uniform grid sizes, dependent upon actual objects in our world.

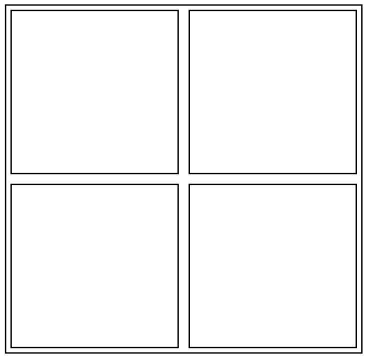


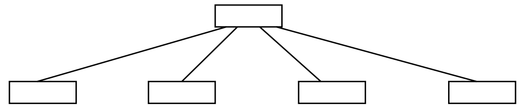
11-19: **Quadtree**

11-20: **Quadtree**

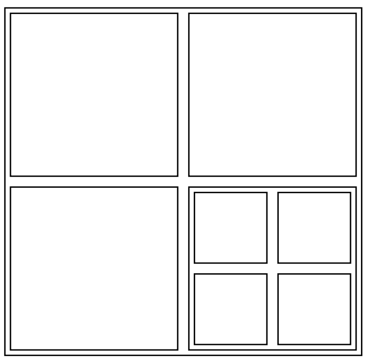
- Tree data structrue
  - Root of the tree represents the entire world
  - Four subtrees, one for each quadrant of the world

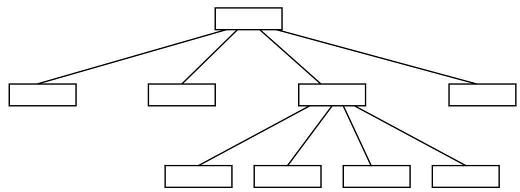
- Each quadrant can be divided into 4 as well





11-21: **Quadtree** 11-22:



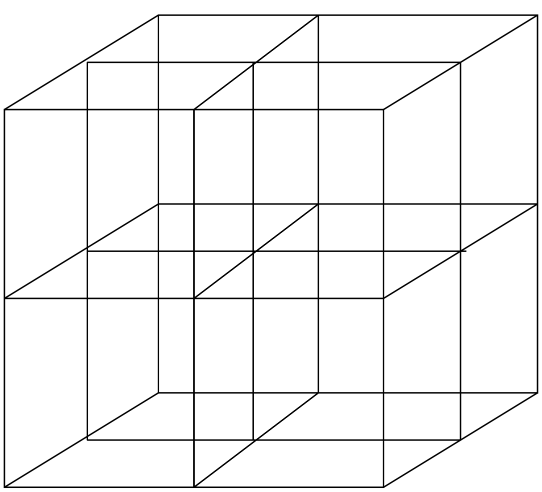


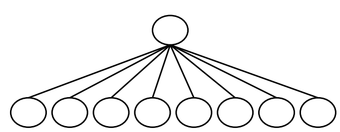
Quadtree 11-23: **Quadtree**

- How would you extend this to 3D?

11-24: **Octree**

- How would you extend this to 3D?
  - Region of space is a cube instead of a square
  - Divide cube into 8 subcubes
  - Result is an Octree





11-25: **Octree** 11-26: **Quadtree**

- Back to Quadtrees:
  - Many different variants of Quadtrees
  - We will be covering a specific varient that stores AABB regions.
  - Operations will be inserting an AABB, finding all elements that intersect a given AABB, and moving / resizing an AABB

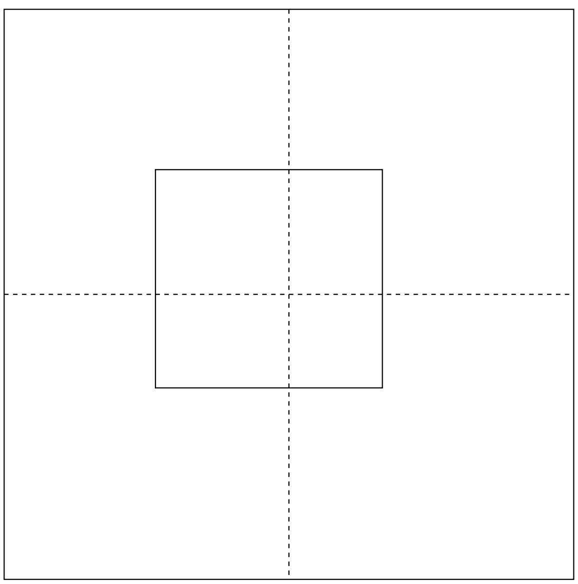
11-27: **Quadtree**

- Quadtree definition:
  - Each non-empty node in the quadtree stores:
    - List of all regions stored at that node
    - Four children, which divide the region into 4 equal quadrants

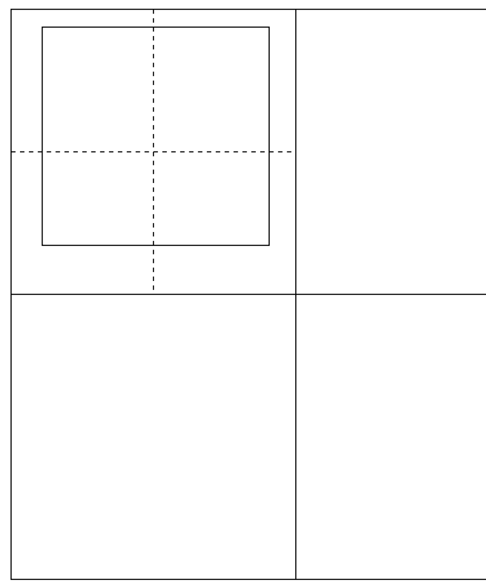
11-28: **Quadtree**

- Regions are stored in the lowest possible node in which they can be completely contained (up to a maxumum tree depth)

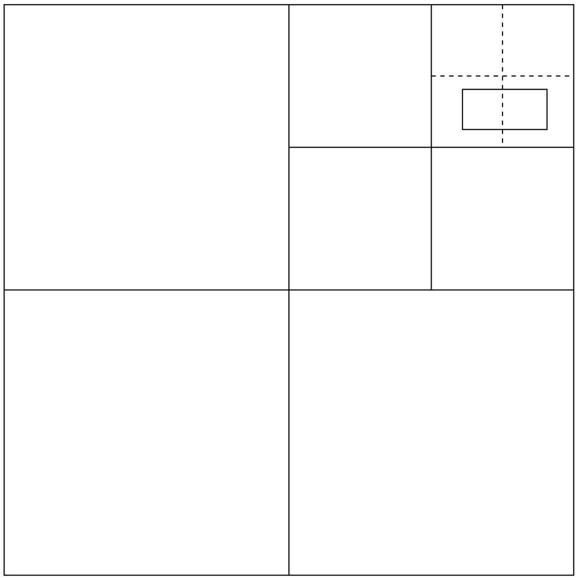
11-29: **Quadtree**



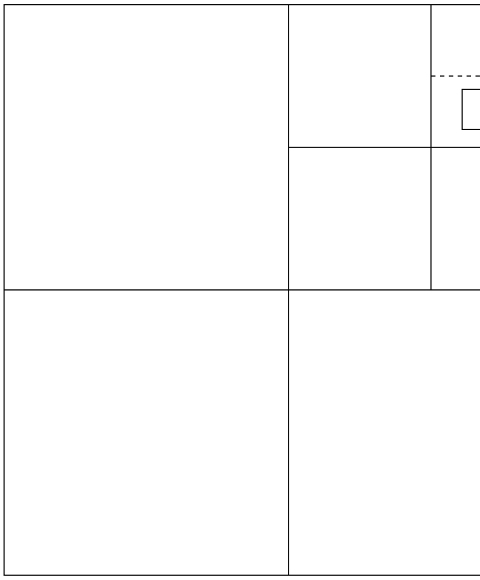
11-30: **Quadtree**



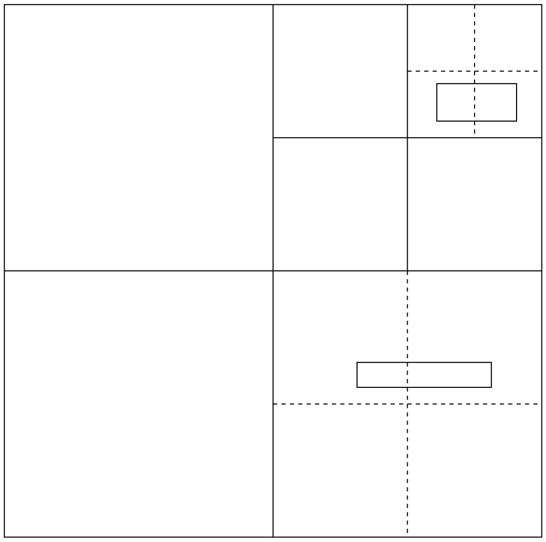
11-31: **Quadtree**

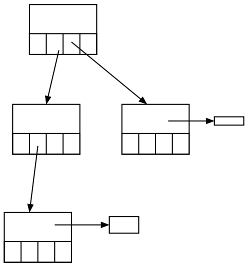


11-32: **Quadtree**

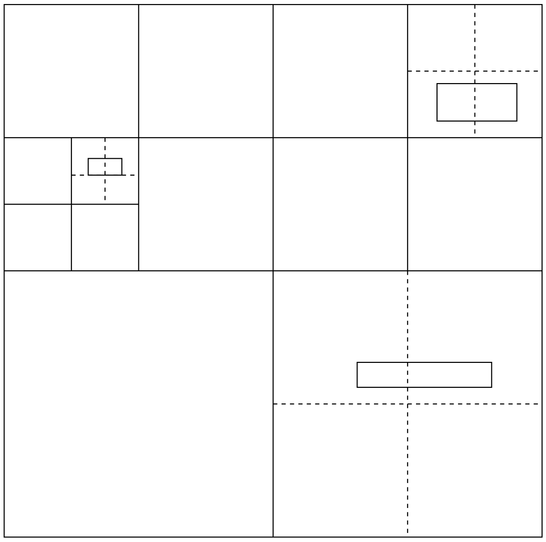


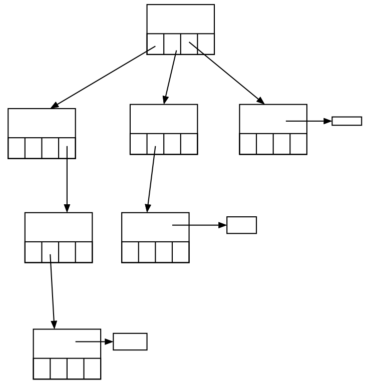
11-33: **Quadtree**





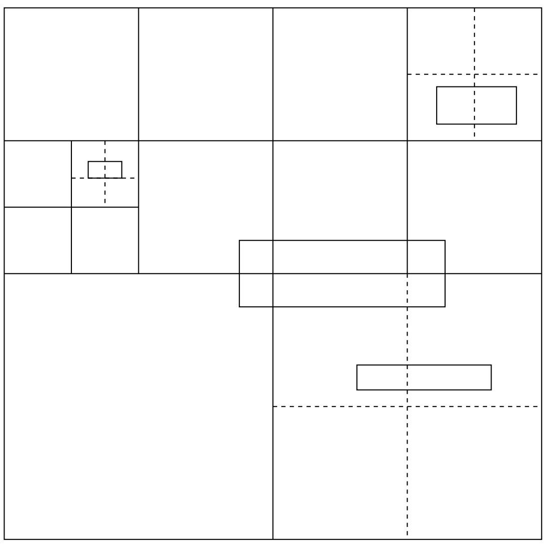
11-34:

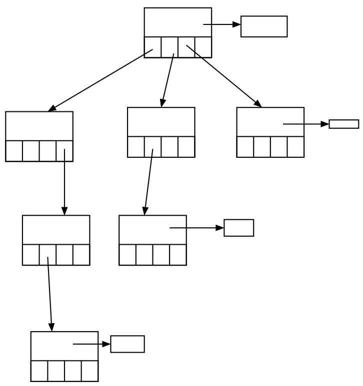


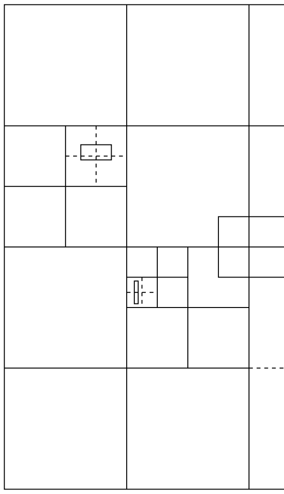


Quadtree

11-35: **Quadtree**

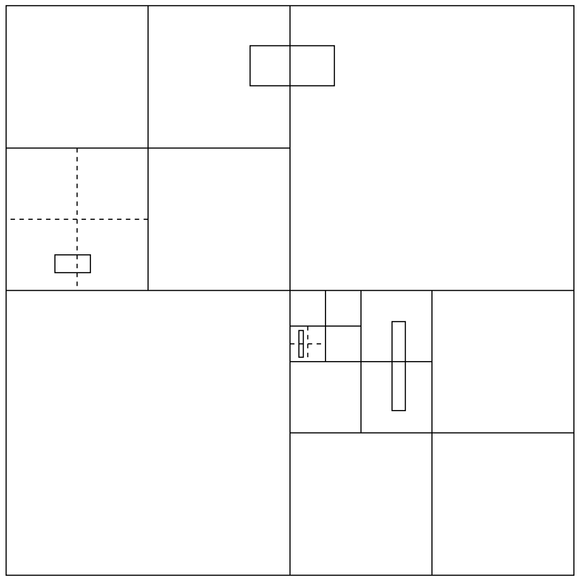




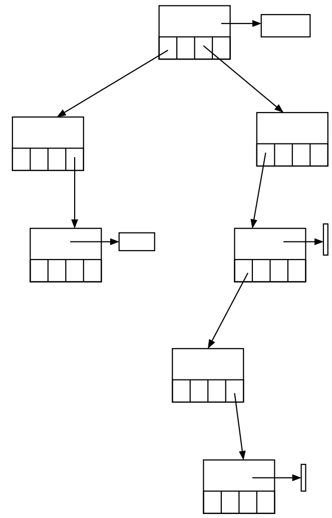


11-36: **Quadtree**

11-37: **Quadtree**

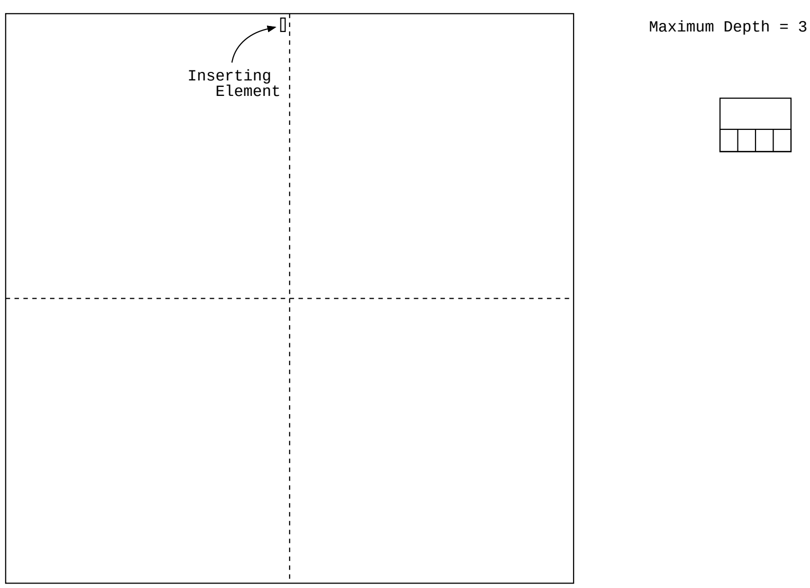


11-38: **Quadtree**

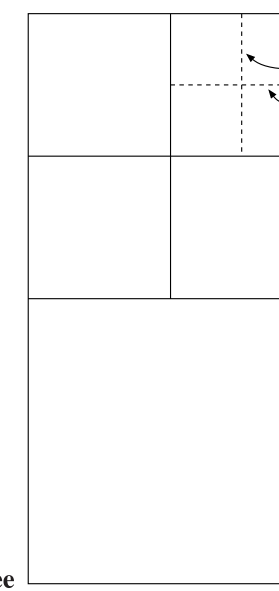


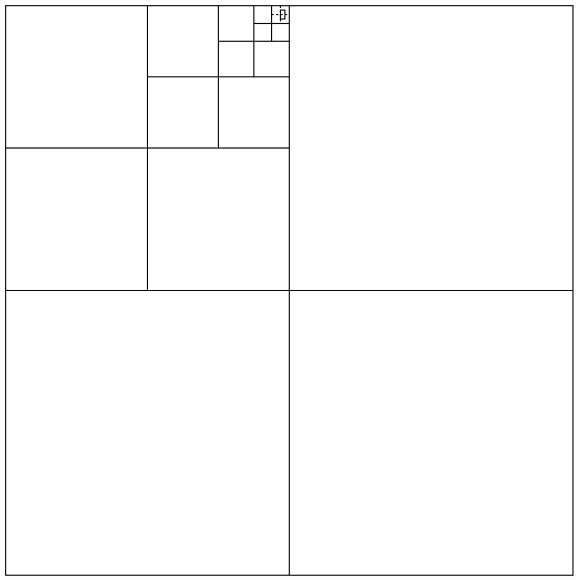
- Maximum Depth
  - We may specify a maximum depth for our quadtree
  - Prevents creating a huge number of intermediate nodes for very small objects
  - We would need to tune the maximum depth for our particular application

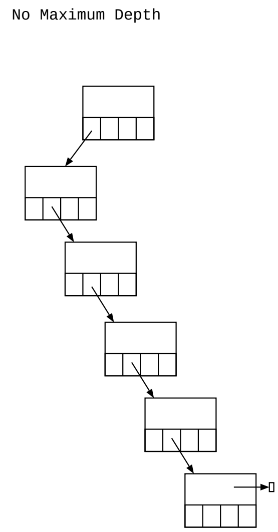
11-39: **Quadtree**



11-40: **Quadtree**



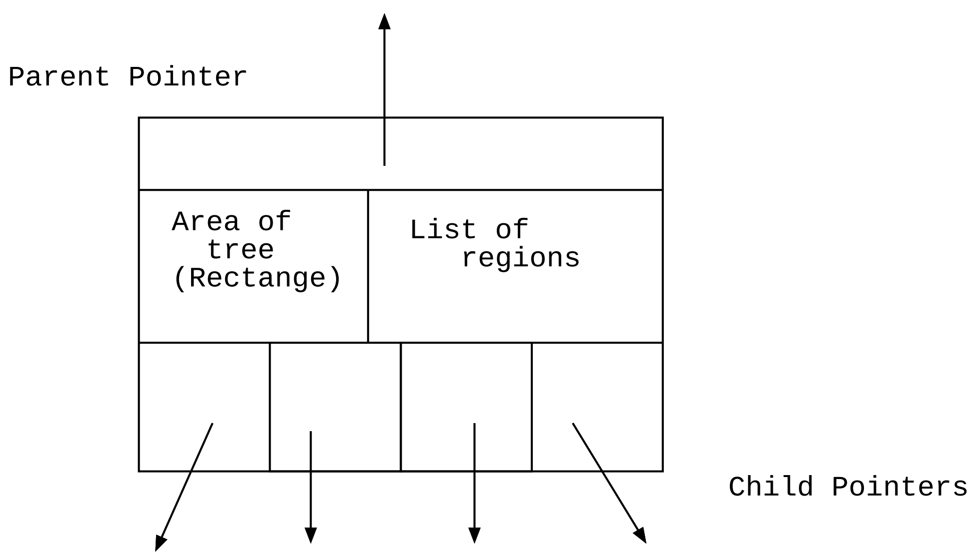




11-41: **Quadtree**

11-42: **Quadtree**

- Implementation: Each Quadtree node stores:
  - Area covered by this node (Rectangle, requires 2 points (or 4 numbers))
  - List of all elements stored in this node
  - Four child pointers (any of which could be null)
  - Parent pointer
    - Don’t need for inserting / finding elements
    - Come in handy for moving, resizing elements



11-43: **Quadtree**

11-44: **Quadtree**

```
public class WorldElem
{
    public Rectangle AABB { get { ... } }
    // Other stuff ...
```

```
}
public class QuadtreeNode
{
    public Rectangle Area { get; set; }
    public ArrayList<WorldElem> mWorldElems;
    public QuadtreeNode Parent { get; set; }
    public QuadtreeNode UpperLeft { get; set; }
    public QuadtreeNode UpperRight { get; set; }
    public QuadtreeNode LowerLeft { get; set; }
    public QuadtreeNode LowerRight { get; set; }
}
```

11-45: **Intersecting**

- How would you create a list of all elements that intersect a particular region?

```
void Intersect(QuadtreeNode t, Rectangle region,
               ref ArrayList<WorldElem> intersecting)
{
   ...
}
```

11-46: **Intersecting**

- Quadtrees are recursive data structures, manipulate them with recurisve functions
  - Base case for finding intersections?
  - Recursive case for finding intersection?

11-47: **Intersecting**

- Base case for finding intersections:
  - Empty tree – no intersections
- Recursive case for finding intersection:
  - Everything at the root node that intersects the query region
  - Plus result of recursive call to each of the 4 quadrants *that intersect the query region*

11-48: **Quadtree**

```
void Intersect(QuadtreeNode t, Rectangle region,
               ref ArrayList<WorldElem> intersecting)
{
  if (t != null)
  {
    foreach (WorldElem elem in t.mWorldElems)
    {
      if elem overlaps with region
        intersecting.Add(elem);
    }
    if (region overlaps upper left quadrant of t.Area)
        Intersect(t.UpperLeft, region, ref intersecting)
    if (region overlaps upper right quadrant of t.Area)
        Intersect(t.UpperRight, region, ref intersecting)
    if (region overlaps lower left quadrant of t.Area)
        Intersect(t.LowerLeft, region, ref intersecting)
    if (region overlaps lower right quadrant of t.Area)
        Intersect(t.LowerRight, region, ref intersecting)
  }
}
```

11-49: **Quadtree**

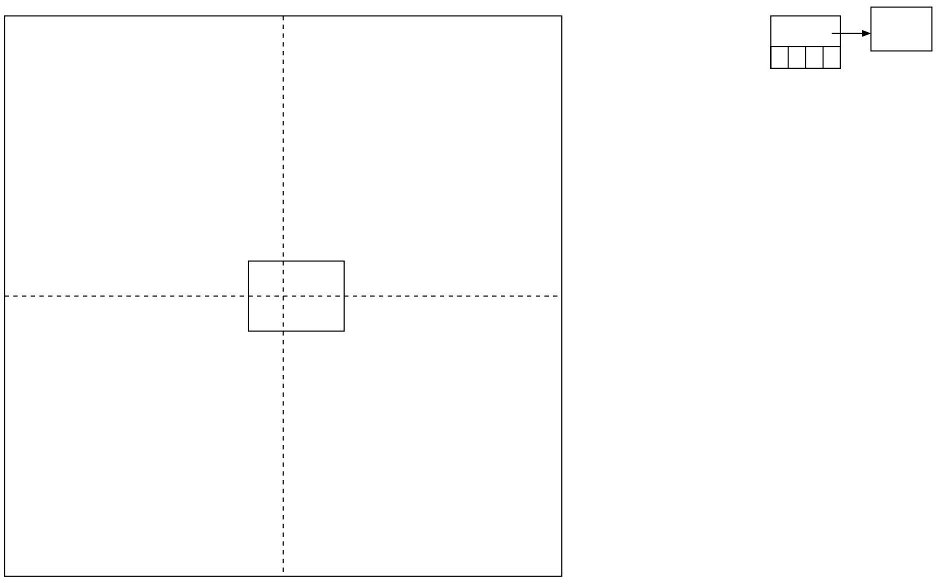
- How do you insert something into a quadtree?
  - Base case / Recursive case

11-50: **Quadtree**

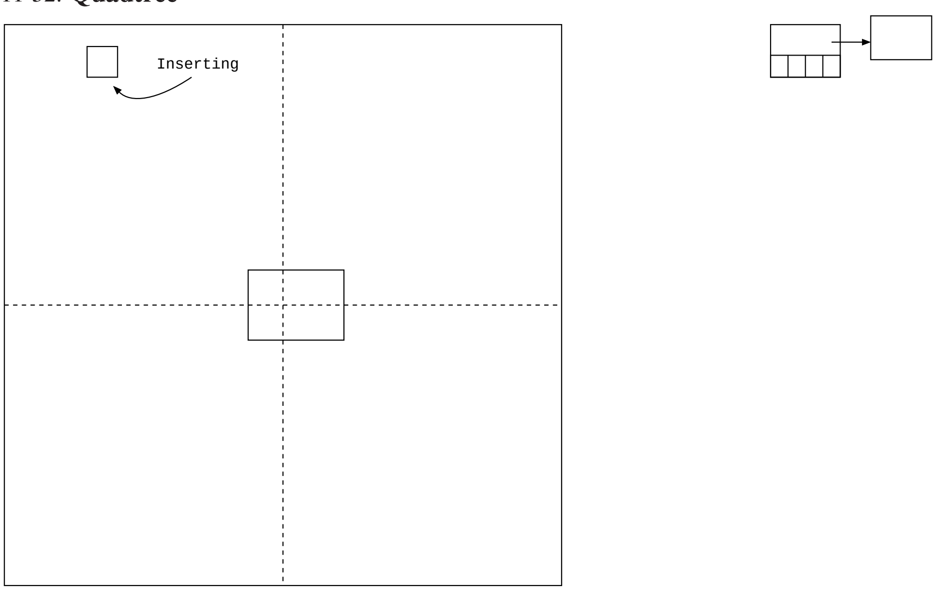
- Wha is the base case – when is it easy to insert an element?
  - Empty tree is not a base case! (why not?)

11-51: **Quadtree**

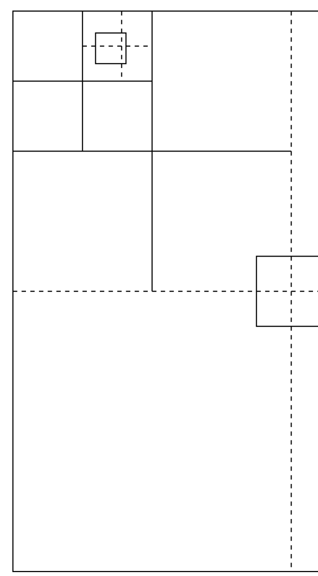
- Wha is the base case – when is it easy to insert an element?
  - Empty tree is not a base case! (why not?)
    - If the tree is empty, may still need to build a great number of tree nodes



11-52: **Quadtree**



11-53: **Quadtree**



11-54: **Quadtree**

11-55: **Quadtree Insertion**

- Base case:

11-56: **Quadtree Insertion**

- Base case:

- Object you are inserting does not fit completely in one of the subquadrents of the current node, or we are already at maximum depth
- Add object to the root list

11-57: **Quadtree Insertion**

- Recursive Case:

11-58: **Quadtree Insertion**

- Recursive Case:
  - Object you are inserting does fit completely in one of the subquadrents of the current node, not at maximum depth
  - Add object to the appropriate subtree
  - May need to create a new subtree

11-59: **Quadtree Moving**

- How can you move an element stored in a quadtree?
  - (That is, when you move an element, how can you efficiently update its position in the quadtree?)
- Assume that you have:
  - A pointer to the object whose AABB has changed
  - A poiner to the quadtree node where this element lives

11-60: **Quadtree Moving**

- If the element is still in the correct location, do nothing
  - How do you know that the element is currently in the correct location?

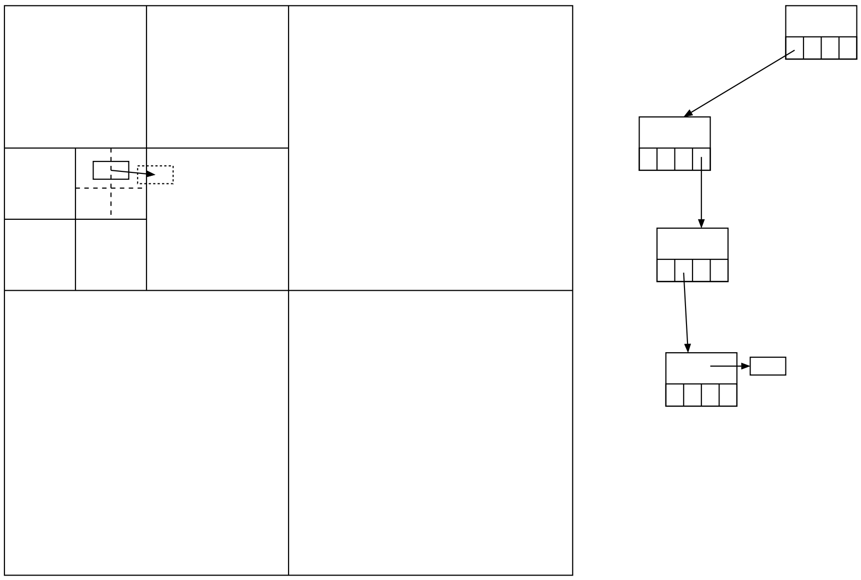
11-61: **Quadtree Moving**

- If the element is still in the correct location, do nothing
  - How do you know that the element is currently in the correct location?
    - The AABB of the relocated node still fits within the region of the node where it lives
    - The AABB of the relocated node does not fit completely within the region of any of the 4 quadrants (or the current node is already at the maximum depth)

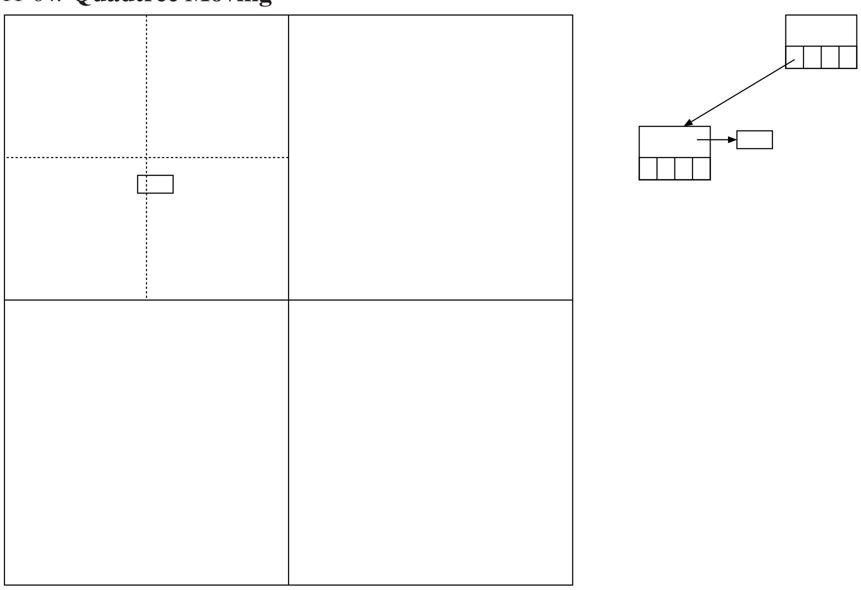
11-62: **Quadtree Moving**

- If the element is *not* at the correct location, where could it be, relative to the node where it currently lives?

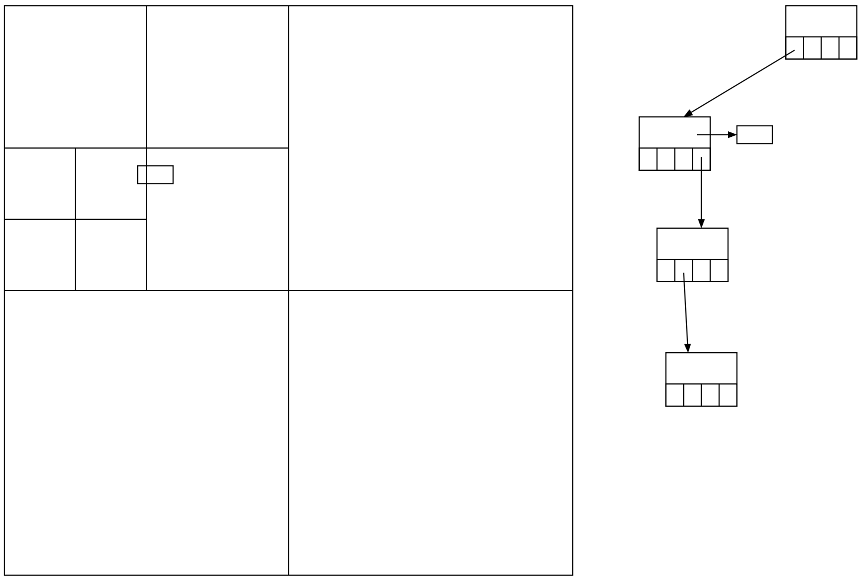
11-63: **Quadtree Moving**



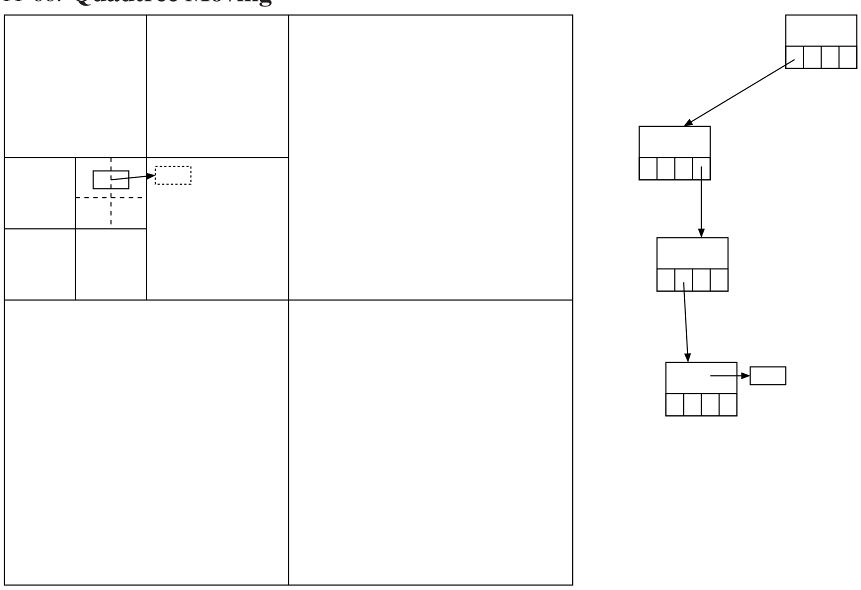
11-64: **Quadtree Moving**



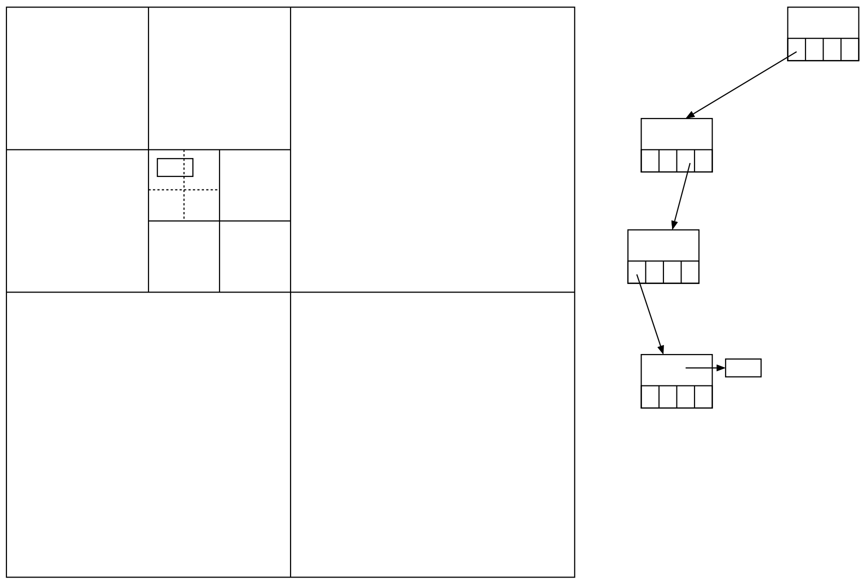
11-65: **Quadtree Moving**



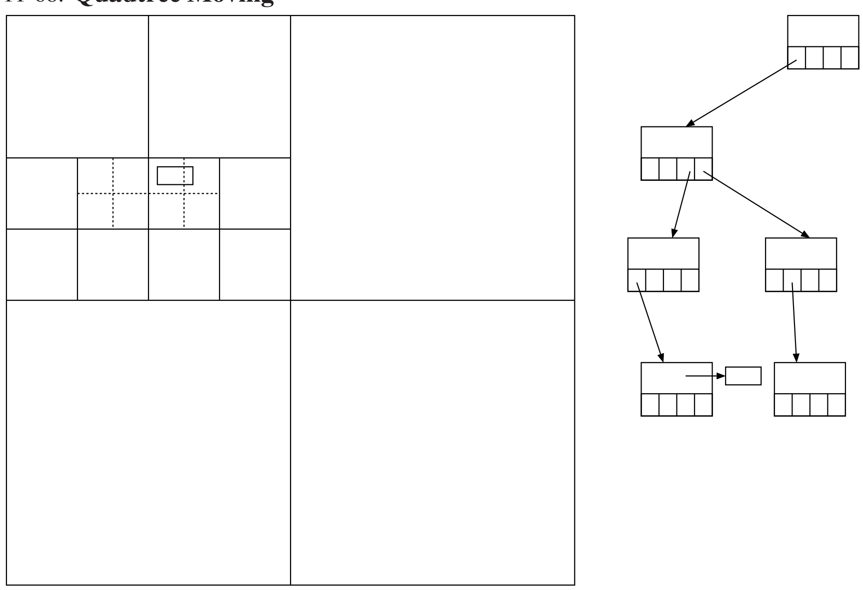
11-66: **Quadtree Moving**



11-67: **Quadtree Moving**



11-68: **Quadtree Moving**



11-69: **Quadtree Moving**

- When an element has moved ...

11-70: **Quadtree Moving**

- When an element e has moved from the note t:
  - Remove element from the list at node t
  - While t doesn't contain e
    - t = t.parent
  - While e fits completely within one of the 4 quadants of t:
    - t = t.quadrant // quadrant that e fits completely inside

    - Insert e in list at t

11-71: **Quadtree Moving**

- When an element e has moved from the note t:
    - Remove element from the list at node t
    - While t doesn't contain e
        - t = t.parent
    - Insert e into tree rooted at t

11-72: **Quadtree Moving**

- What if the AABB for an object doesn't just move, but changes
    - Object rotates, for insance

11-73: **Quadtree Moving**

- What if the AABB for an object doesn't just move, but changes
    - Object rotates, for insance
- Exact same code will work
    - Move up until you reach a node that completely contains the object
    - Move down until you reach a node that just barely contains the object (won't fit in any children of node)

11-74: **Quadtree Moving**

- Moving nodes may create empty subtrees
    - Tiny object moves across the entire world
- Should we clean up the tree, removing unused nodes?

11-75: **Quadtree Moving**

- Moving nodes may create empty subtrees
    - Tiny object moves across the entire world
- Should we clean up the tree, removing unused nodes?
    - It depends!

11-76: **Quadtree Memory**

- If we don't prune the tree, we could run out of space
    - Small object moving all over the world
- If we do prune the tree, lots of calls to a memory manager
    - small object moving around in a smallish region

11-77: **Quadtree Memory**

- If we never remove empty subtrees when elements move, eventually the tree will be complete
  - No more calls to new
- Could also “Prefill” the tree
- Problems with this method?

11-78: **Quadtree Memory**

- If we never remove empty subtrees when elements move, eventually the tree will be complete
  - No more calls to new
- Could also “Prefill” the tree
- Problems with this method?
  - Could use too much memory
  - Solution: Limit depth of the tree