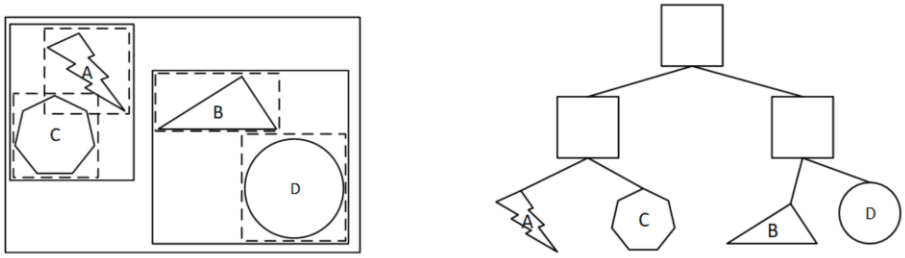
Dynamic Aabb Trees

jodavis42@gmail.com

Bounding Volume Tree

A hierarchy of bounding volumes

Typically in a binary tree



Another common kind of spatial partition is a bounding volume tree. Bounding volume trees are simply hierarchical trees where each parent node in the tree has a bounding volume that contains all of the children. At the root level, the bounding volume represents all of the objects in the tree.

BVTs can be constructed with any kind of a bounding volume, but aabbs are one of the most common (and best) bounding volumes for this. With them being nested in a tree structure we can reduce the asymptotic complexity from $O(n^2)$ to $O(\log(n))$.

One of the best things about BVTs is that they are very versatile. They work well for casts and pair queries. They are easy to insert and remove objects. They're also easy to implement. In fact, after implementing an n-squared bounding volume spatial partition I recommend doing a BVT before any other spatial partition.

It's also worth nothing that typically all internal nodes are empty and leaf nodes contain all data. Also it is common that a leaf node contains only one object.

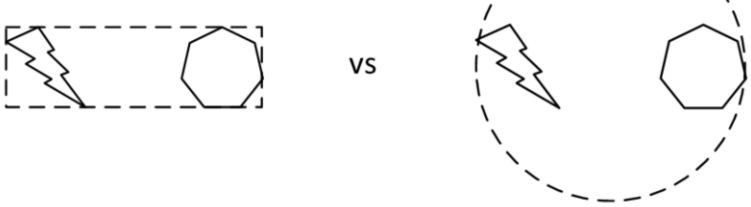
Why Aabb Trees?

Cheap to compute

Cheap to update

Cheap to intersect

Tend to take less volume than spheres



So this begs the initial question, why aabbs? Well for cheap intersection tests aabbs and spheres can't be beat. Aabbs are cheaper to compute for an object which is also a plus. But the biggest factor is that aabbs tend to stack in a hierarchy better than spheres. Another way to put it: an aabb of two aabbs tends to be smaller than a sphere of two spheres. Of course an obb would be even better than an aabb, but computation and intersection costs tend to make this not an option.

Dynamic Trees

Focus on dynamic trees

- Incremental insertion/removal

Discuss static building later

For the most part I'll focus on dynamic tree building. We typically have games where objects are created and destroyed often so we need to have some way to incrementally insert/remove objects in the tree.

That being said, I will talk later about some static tree building techniques. Static building tends to produce a more tight-fit tree, but at a very large cost. This can still be useful for static mesh mid-phases and even sometimes on the first build of a scene. Also, statically built trees can be updated just the same as dynamically built ones. Dynamic updates are far easier to understand though so I'll start with that.

Dynamic Tree Node

```
class Node
{
  Aabb mAabb;
  void* mClientData;

  Node* mLeft;
  Node* mRight;
  Node* mParent;
  size_t mHeight;
};
```

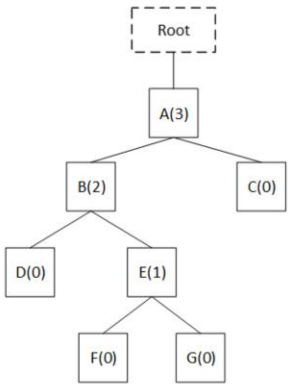
Before talking about any algorithms it's first important to establish what data we need on the node. Obviously a node needs its aabb. We also need to store the client data for all leaf nodes. All internal nodes also need the left and right pointer (this is a binary tree after all). At this point we could keep a flag to signify if the node is internal or leaf but this can be derived from the left or right pointer.

The final two pieces of data are not needed in all tree representations but for the dynamic aabb tree we need them. The first is the parent pointer, as we'll be re-linking nodes so traversing up is necessary. The other is the height of the node.

Tree Height

The height of a leaf node is 0

An internal node has a height of one more than its largest child

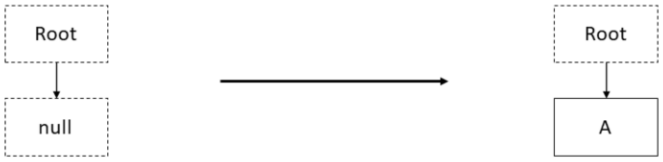


*Height won't be important until we get to balancing

The height of a node won't be important until later when we re-balance the tree. The important thing to know about the height now is that all leaf nodes are height 0. Any internal node's height is simply calculated from its children (a parent is always one higher than it's highest child). This means that the height grows as we traverse up to the root.

Insertion

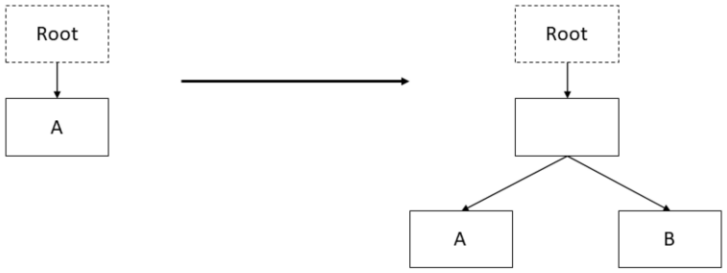
Case 1: Empty Tree



Since we're doing incremental insertion into a tree the first case we have to worry about is when inserting into an empty tree. This is straightforward as the node just becomes the root.

Insertion

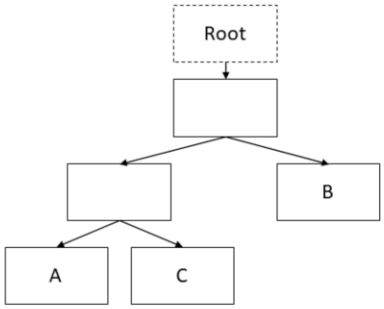
Case 2: 1 Node



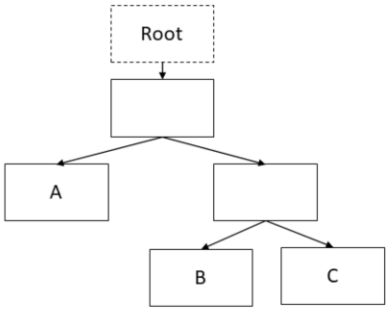
The second case to worry about is also very easy: the case when there's only 1 node in the tree. In this case we simply create a new node to be the parent of the two leaf nodes and then link them all up. It's typically not important who becomes the left or the right child, although for this class we'll always make the old node the left and the new node the right (for unit testing). Also note that we need to update the height and aabb of this new node.

Insertion

Case 3:



Which side do we choose?



Now we actually have to decide on something. When we insert a new node how do we know which side to send our new node down?

Node Selection

Add a node selection function

```
Node* SelectNode(Node* insertingNode, Node* node0, Node* node1)
{
  return ??;
}
```

At this point we can introduce the concept of a node selector function. Simply put, this function will choose which side of the tree we should traverse down given the new node we're inserting.

There are quite a few methods to determine which node to remove.

Node Selection - Height

Select the smallest sub-tree

```
Node* SelectNode(Node* insertingNode, Node* node0, Node* node1)
{
  if(node0->mHeight < node1->mHeight)
    return node0;
  return node1;
}
```

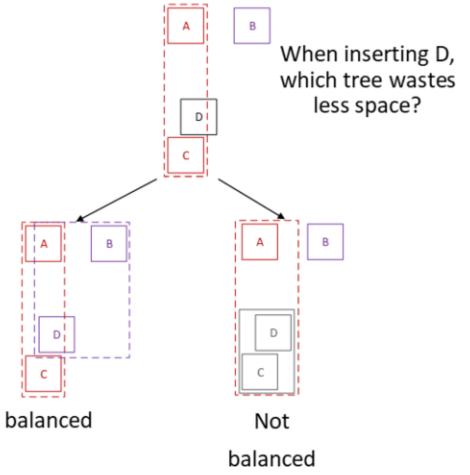
Is this a good heuristic?

We could choose to use the height of each sub-tree to determine which side to go down. This will help us keep our tree balanced, but what are the ramifications of this?

Node Selection - Height

Height is a bad heuristic

We need to take into account the aabb sizes



If we look at a small example with selecting based upon height it should become apparent that height isn't a good metric. The kind of queries we'll be performing on the tree will rely on geometry queries, not height queries. If we produce a tree that is more likely to have false-positives then we will be wasting computation. This should lead us to realize that a good tree wastes as little space as possible.

Node Selection - Distance

Choose closest aabb (by center)

```
Node* SelectNode(Node* insertingNode, Node* node0, Node* node1)
{
  float dist0 = Distance(insertingNode->mAabb, node0->mAabb);
  float dist1 = Distance(insertingNode->mAabb, node1->mAabb);
  if(dist0 < dist1)
    return node0;
  return node1;
}
```

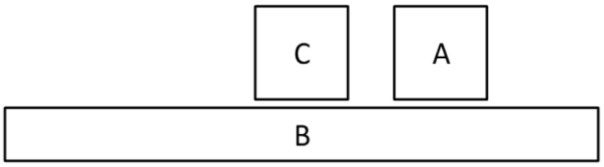
Does this heuristic fail anywhere?

The simplest reasonable heuristic is now distance. If we group two aabbs that are close by together then we mitigate the risk of wasting a lot of space.

This leads us to the simple question, how can this metric fail (produce a really wrong answer)?

Node Selection

Distance is a poor approximation of size growth



We want some metric of size delta

The distance metric fails by not taking into account the amount of space wasted. In the above example, Node B is closer to C than A so they'll be grouped together. This should obviously be a bad grouping as it wastes significantly more space than if C and A were grouped.

This leads us to realize we want some metric that tries to account for the change in a node's size.

Node Selection - Volume

How about choosing the minimum volume increase?

Smaller volume -> Less wasted space

Does this heuristic fail anywhere?

Since we want to minimize wasted space this naturally implies that the volume (or area in 2d) of the aabb is a good metric. By minimizing the volume increase we minimize how much space we waste at each step which should logically minimize the chances that a collision test would return a false positive.

Once again, where does this heuristic fail?

Node Selection - Volume

Unfortunately…

Grouping by volume doesn't prevent long thin volumes

Unfortunately volume isn't the right metric to use here as A and C will be grouped because the volume is less than grouping B and C. Luckily the fix for this is not hard.

Node Selection – Surface Area

Choose the smallest increase in surface area
- Surface area grows with volume
- Also deals with long-thin objects

Best for most query types (Aabb, Ray, Frustum, etc…)

Does this heuristic fail anywhere?

The standard heuristic to use is surface area. As an objects volume increases so will the surface area, however the surface area will also be large for long thin objects. Surface area heuristics are typically used for ray-casting, however they also make sense for any form of a volume test.

Note that when I say surface area I refer to the small increase in surface area, not just the smallest surface area. We want inserting into a sub-tree with a small aabb that doubles in to be worse than inserting into a sub-tree with a large aabb that doesn't increase in size at all.

Once again, does this heuristic fail anywhere?

Extra Heuristic Considerations

Surface area doesn't really fail anywhere, but there are some additional considerations

Is a node a leaf or internal?

Tree height?

Etc…

Note: These are not part of your assignment

We could start trying to consider more things than surface area to build a better tree. What about height? Should we include how much we're imbalancing a tree in the cost? What about splitting a node that was a leaf vs. recursing down an internal node? At this points heuristics start to look like: $Cost_A = \mathrm{T_{SA}} * \Delta SA + T_{split}$, that is some weight is assigned to surface area and another weight is assigned to the cost of splitting child nodes vs. going down internal nodes.

Heuristics can become very complicated if we consider these and if we aren't careful we might spend more time computing a heuristic than we gain from building a better heuristic. For this class we'll stop at simply using the smallest increase in surface area.

Insertion – Final Remarks

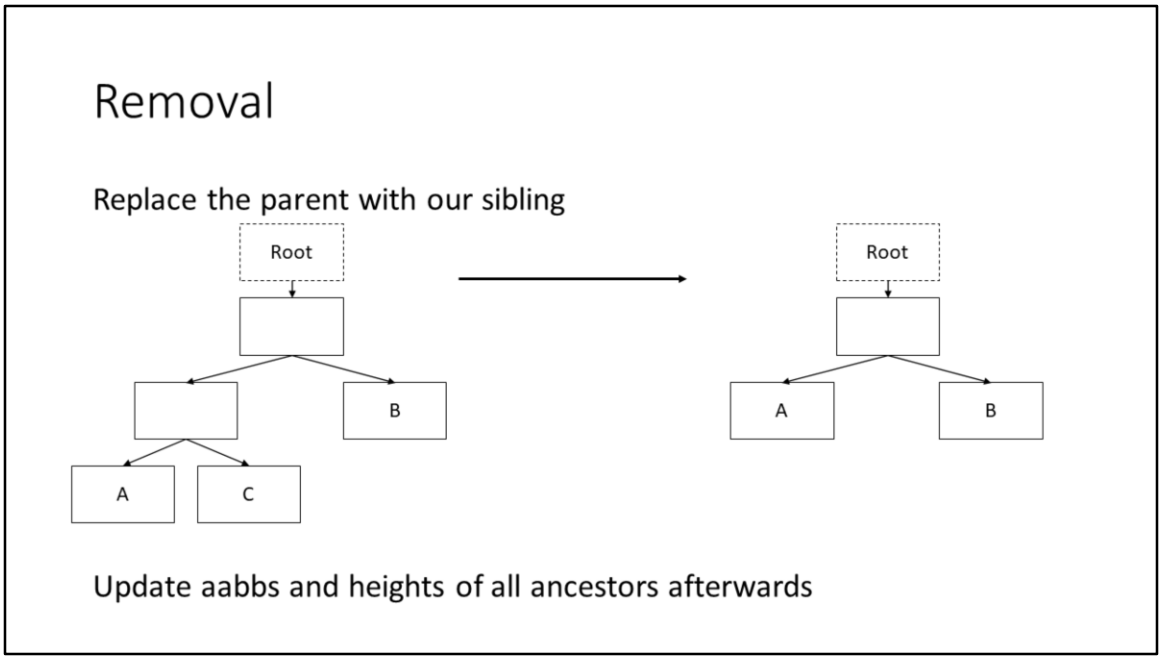
Keep selecting nodes until you reach a leaf node

Do a pass back up to update aabbs and height

*Heights cannot be updated while recursing down

Now that we have defined our node selection function we can finish up describing insertion. Simply select nodes until we reach a leaf and then split (old node on left). At this point we have inserted the new node into the tree, but the heights and aabbs are out of date. Now we can do a simple pass up the tree that will fix these values.

There's two things to note about updating these. The first is that this should be a $\log_n$ pass back up the tree. Your algorithm should in no way be n or n^2. The second thing to note is that while we can update aabbs on our way down the tree, we cannot update heights. It is entirely possible that a node's height will not be increased even though a node is being inserted below it.



For the most part removal is easier than insertion. As long as we can quickly find the node we're removing (using proxies should make this a constant time operation) then we just have to link up our sibling and grand-parent and then delete ourselves and our parent.

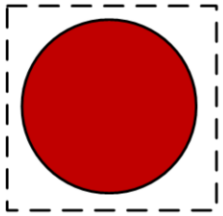
The only special cases here are when we don't have a grand-parent, in these cases we just have to properly link back up the root.

Once again we also need to walk up the tree, this time from the sibling and update the aabbs and heights of each node.

Update

Remove then re-insert

To avoid constant updates use a fat aabb



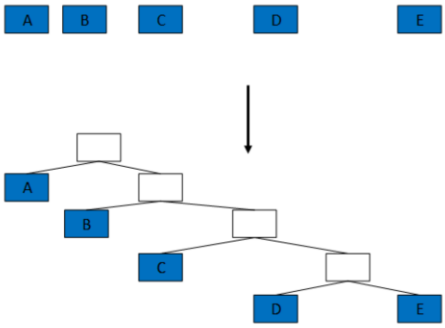
Allows wiggle room

Unfortunately there's not really any optimized way to update a node in a tree if it moves outside of its bounding volume. There is one common trick used though to reduce constant updates: fat aabbs. As we only have to remove and re-insert a node if we are no longer contained in the bounding volume we can artificially inflate the aabb for every node inserted. This way if an object slightly moves or rotates then no update will happen.

Fattened aabbs typically are just computed as a scalar on the actual aabb, but sometimes completely different methods are used. The most common other method is to sweep out an aabb based upon the objects velocity so that it will ideally contain the object for the next frame.

Balancing

Insertion order can cause an unbalanced tree



Even if we currently use some really good surface area heuristic, there's a chance we can create a fairly unbalanced tree. In particular, this is controlled by our insertion order. We may insert objects in such a way where the best heuristic causes a completely unbalanced tree.

One way to deal with this is to randomize insertion order, but as the tree is dynamically added to we might not have control of this. Because of this we need some method to forcibly re-balance the tree.

Balancing - Randomization

Randomly pick a node and re-insert it

Heuristic should produce a more balanced tree

Problem:

Can take a while to balance out

One method to rebalance the tree is to use our update to effectively create a random insertion order. By picking random nodes in the tree (not pure random but using some logical way to iterate through the tree) to re-insert our heuristic should pick a better insertion spot.

Unfortunately, this can take a very long time to balance out a bad tree so we can have a lot of frames of bad performance until then. Instead we can try a more direct approach.

Balancing – Tree Rotations

Fix any unbalanced sub-trees (using height)

1. Identify pivot, small child, and large child
2. Detach small child, pivot, and old parent
3. Replace the grand-parent link of the old parent to the pivot
4. Insert old parent as new small-child
5. Insert small child where the pivot node was

*Make sure to use this algorithm for your assignment

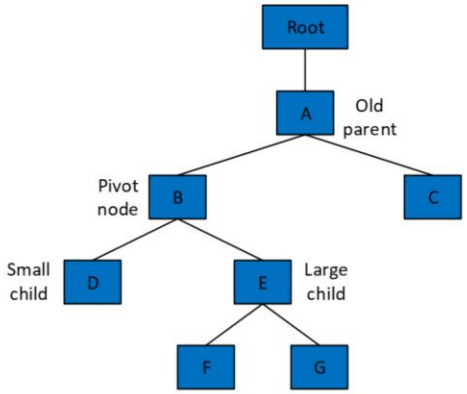
The other method is to force the tree to be balanced by keeping track of tree height. We can insert a node as normal using our heuristics and then if we encounter any un-balanced sub-trees we can rotate them so they are balanced. An tree rotation is just 5 simple steps. Hopefully the next pictures help to explain it.

Do note that we only fix an un-balance of more than 1.

In some previous semesters students have tried to use other algorithms instead of the one presented here. While it is true that there are other ways to rotate a tree, this one preserves more sub-trees than most other methods. Make sure to use this algorithm for your assignment otherwise you will not pass the unit tests!

Tree Rotation

1. Identify the pivot and its small and large children

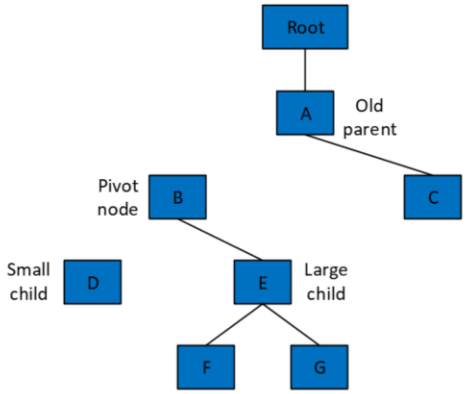


When we manage to find a node that is imbalanced (that is the height difference between its 2 children is more than 1) then we have to rotate. In this case we'll notice at node A that B has a height of 2 while C has a height of 0. The idea then is to "rotate up" node B to take the place of A.

The first step is to label the larger of the children here the pivot node. The pivot node is the one that will rotate up and replace its parent in the tree. After this we need to identify the small and large child of the pivot. This is important so that the new sub-tree we create will be balanced. If these nodes are tied then we have to just pick one (see your assignment document for which one).

Tree Rotation

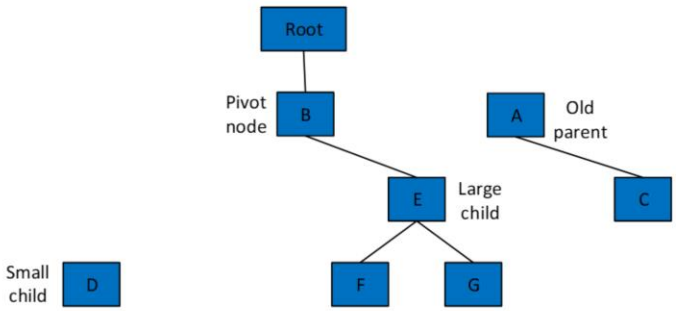
2. Detach small child from pivot and pivot from the old parent



Now we can split our tree into 3 different sections: the old root (our parent), the small child's sub-tree, and then the pivot node (still connected with the larger child). We will re-link these sub-trees so that the entire sub-tree becomes balanced.

Tree Rotation

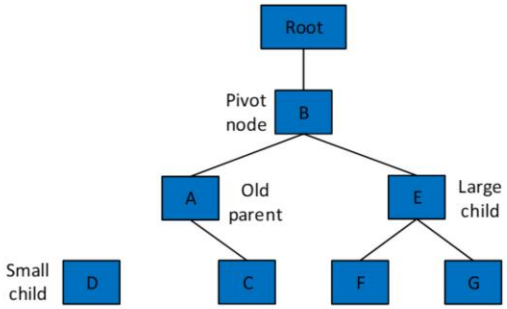
3. Replace the grand-parent connection of the old parent with the pivot



As we are “rotating up” the pivot node (B), we need to detach the old parent from the pivot node’s grand-parent and then replace this connection with the pivot node. Make sure that the pivot node is put on the same side as the old parent. Also be careful in the case where the old parent was the root (the grand-parent was null).

Tree Rotation

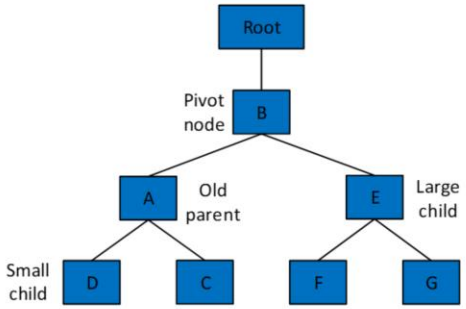
4. Attach the old parent under the pivot where the small child was



Since we rotated B up this means that A has to rotate down. As we left the old large child attached to B we need to attach our old parent (A) on the side were we removed the small child from.

Tree Rotation

5. Attach the small child to the old parent where the pivot node was



Finally, we need to re-link up the small child. Our old parent used to point to the pivote node as either the left or right child. We need to take this link and make it point at the small child from before.

Now with that done we should have a balanced sub-tree.

Tree Rotation

Need to re-balance on insert and remove

Check all sub-trees from affected point

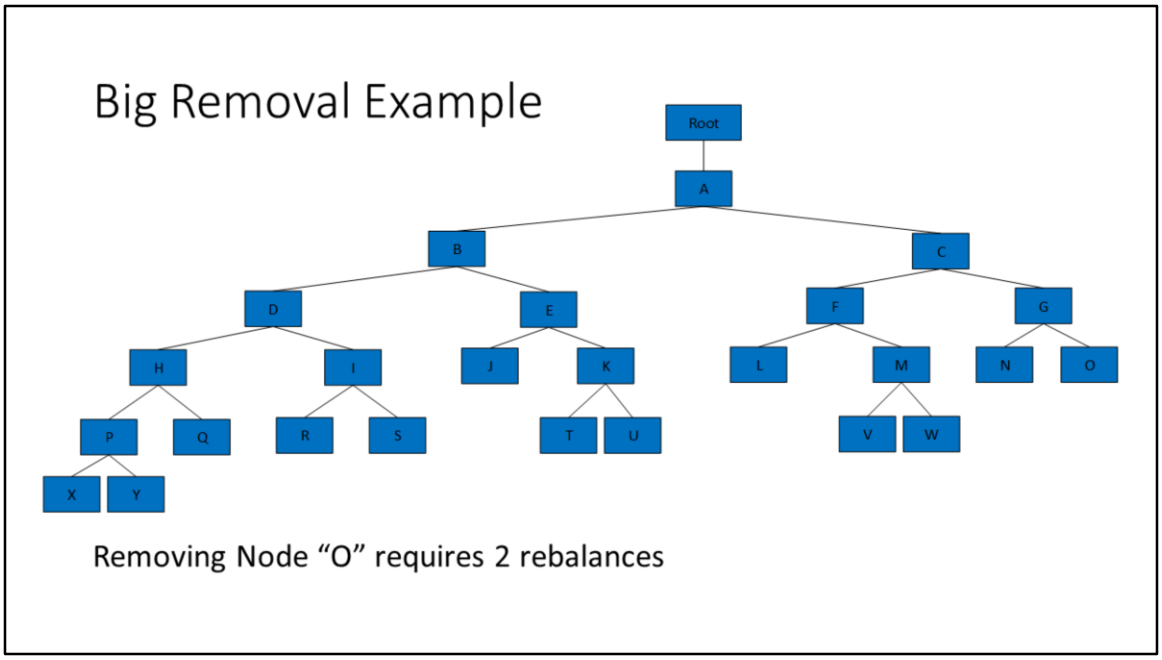
Make sure to update aabbs and heights

We obviously need to check for a re-balance on insertion and removal, but what exactly do we need to check for re-balance.

Well on an insert we ended up at a leaf node. That leaf can't be unbalanced so just check the parent and above (the parent and grandparent shouldn't ever be unbalanced but we need to update their aabbs and height anyways). We have to check for an unbalanced node at each level because we could have various unbalanced sub-trees.

Removal is similar, we need to check the old grand-parent and above for re-balance.

Do note that we need to update the aabb and heights after any rotations!



This is an example of a removal that can result in more than one rotation in the tree. You must continue up the tree checking for any imbalanced nodes until you reach the root!

Rotation vs. Randomization

Tree rotation tends to perform better

Unfortunately balanced isn't always better

That said, tree rotation is good enough

So which is better between tree rotations and randomization. Well the rotations will definitely fix any issues much quicker than randomization. Also if a tree is already balanced then rotations are faster. That said, tree rotations check anytime an update is performed where randomization only happens a few times per frame. Hence all insertion/removals are a little slower with rotations while randomization is a fixed overhead.

There is one more interesting thought to note though, is a balanced tree better? Technically we don't really care about balance due to height, but rather surface area, but at the same time we can't have a completely imbalanced tree (by height) otherwise we just have a list. However there are some cases where a huge imbalance is better, namely with large size disparities. If we have one really large object it would be good if it sat near the root and didn't inflate any other nodes. By relying on the surface area heuristic we're likely to have this happen.

To be honest, I don't have an answer on this. I personally have both implemented and physics uses randomization while graphics uses tree rotations.

Casting

If the casted shape overlaps a node's aabb, recurse down both sides

Performing a generic shape cast is very simple. At each given step just check if the query shape overlaps the node's aabb (or check the query shape's aabb). If it doesn't then return, otherwise you recurse down both children.

Frustum Casting

If an aabb is outside the frustum then return (no recursion)
If an aabb is full contained then add all children (no more checks)
Otherwise recurse

How well does this prune and can we do better?

Frustum casting can be specialized for an aabb tree to get better performance. The base case is when a frustum intersects a node's aabb, in this case we have to recurse into both children until we hit a leaf node. If we hit a leaf then we add the client data as a result. The more interesting cases come when the frustum doesn't intersect or completely contains a node's aabb.

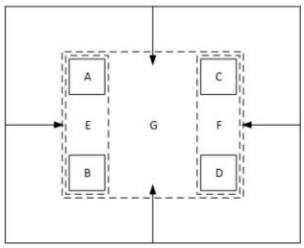
If a frustum doesn't intersect a node's aabb then there's no point in checking children nodes, they are by definition outside the frustum as well. No more recursion or checks of any kind should happen with the node's sub-tree. This speeds up queries by rejecting large sub-trees that are pointless to check.

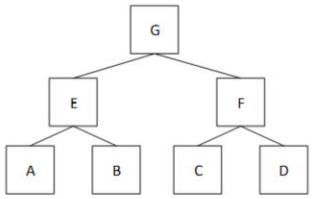
The other important case to consider is when a frustum full contains a node's aabb. In this case all child node's are also completely contained within the frustum due to how the tree is constructed. In this case all leaf children of this node should be added as a result with no further aabb tests.

Now we can look at a few examples to determine how much we prune and if we can do any better.

Frustum Casting Examples

How many FrustumAabb tests will this need?





Tree Hierarchy

How about PlaneAabb tests?

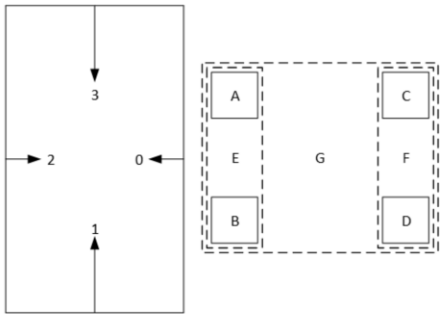
The simplest example is one where the frustum completely contains the tree.

On the right is a tree hierarchy. To the left is the geometric representation of this tree. The box with arrows pointing in is the frustum we are testing against our tree. So in this case how many FrustumAabb tests are required?

The frustum should test against node G and determine that it is completely contained. Since this node is completely contained we should add all children nodes with no further tests, resulting in exactly one FrustumAabb test. It becomes important to look at PlaneAabb tests as well. As a frustum test is 6 plane tests, to get results will also take 6 (for a 3d frustum) PlaneAabb tests.

Frustum Casting Examples

How many FrustumAabb/PlaneAabb tests will this need?

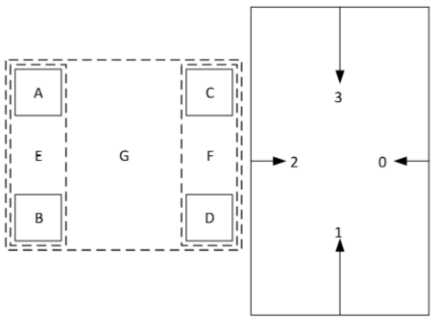


For a more interesting example lets look at a case where the root node is completely outside the frustum. I've numbered the frustum planes here so we can determine the order we check them in.

As the root node is completely outside of the frustum we should terminate again with only 1 FrustumAabb test. Now what about plane tests? Since the root is outside plane 0 we can actually terminate after only one PlaneAabb test! We got the best performance we could have hoped for!

Frustum Casting Examples

How many FrustumAabb/PlaneAabb tests will this need?



Now we'll perform the same test but put the frustum outside on the right of the root. Note that in this case we still terminate after only one FrustumAabb test, but it'll take 3 PlaneAabb tests to determine that the root is outside the frustum.

Frustum Casting – Frame Coherency

Order of plane traversal affects how many PlaneAabb tests are required
What if we can make a good guess for the best plane?

Use coherency: Objects don't move around much
Last frame's axis is a good guess!

This is unfortunate as we could take anywhere from 1 to 6 plane tests to early out. If we simply get lucky we could save a lot of tests. What if we could somehow make an educated guess that would almost always start with the plane that would return outside?

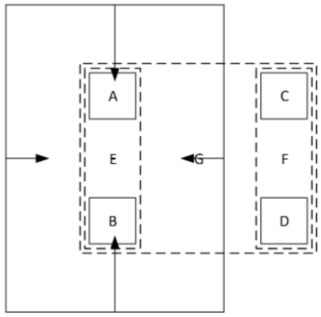
We can actually do this surprisingly easily. Games tend to have a high amount of coherency, that is objects don't move around much from one frame to the next. This leads to the observation that if a node was outside of a particular plane last frame then it's likely to be outside that same plane this frame!

So how can we actually implement this? Luckily it's quite simple, just store that last plane index that a node was outside of on the node itself. If this index is still the one we're outside of then we'll early out right away. If it's not then we may have to check all 6 planes, but we would've had to do that anyways. The only downside is we have to store more memory.

With this, we can assume that we'll perform the minimum number of tests necessary to collect all nodes in a tree.

Frustum Casting Examples

How many FrustumAabb/PlaneAabb tests will this need?



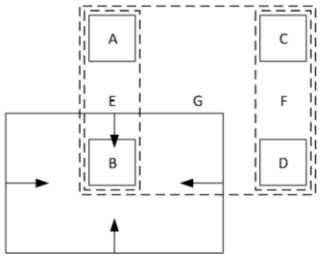
Now we can do the last few examples assuming that our last axis was correctly cached. This is a slightly more complicated test so let's break this into pieces:

We start with node G but we get an Intersection result which takes 1 FrustumAabb and 6 PlaneAabb tests.
Now we recurse, let's first look at F. F is outside the frustum so we should be able to terminate with only 1 FrustumAabb and 1 PlaneAabb test.
Now for node E. E is completely contained within the frustum so it only takes 1 FrustumAabb test. Unfortunately, there's no way to lower the plane tests so we still have a total of 6 PlaneAabb tests required.

This test takes a total of 3 FrustumAabb tests and 13 PlaneAabb tests.

Frustum Casting Examples

How many FrustumAabb/PlaneAabb tests will this need?



For the last example we'll look at the most complicated one.

Node G is an intersection: 1 FrustumAabb and 6 PlaneAabb tests.
Node F is outside: 1 FrustumAabb and 1 PlaneAabb test.
Node E is an intersection: 1 FrustumAabb and 6 PlaneAabb tests.
Node A is outside: 1 FrustumAabb and 1 PlaneAabb test.
Node B is inside: 1 FrustumAabb and 6 PlaneAabb tests.

Total this takes 5 FrustumAabb and 20 PlaneAabb tests.

Ray-Casting

Basic Ray-Casting is just a shape cast:

- If the ray hits an aabb then recurse
- Otherwise don't

This will get all hit aabbs, what if we only want the first?

Note: Do the basic ray-casting for your assignment

Basic ray-casting is very simple. Simply recurse down any node that is hit.

The only other thing to talk about with ray-casting is how you can optimize a raycast that only cares about the first time of impact. This is significantly trickier than one would think.

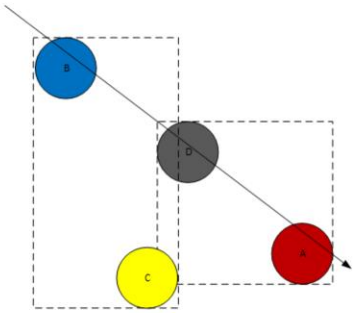
Ray-Casting – First Impact

Idea #1:

For a given node check both children aabbs

Recurse into the first aabb that is hit

If you hit a leaf return

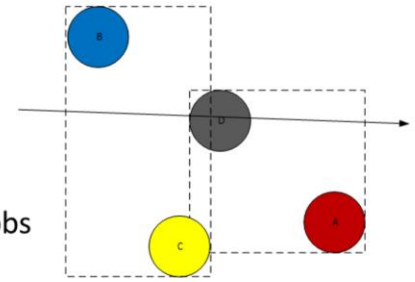


The first idea one might try for an efficient ray-cast is to determine the t-values for when you hit each child-aabb and only traverse into the first one you hit. The algorithm would simply terminate when you hit a leaf node and that would be the first one you hit right?

There's a few problems with this but let's start with the most obvious one…

Ray-Casting – First Impact

Problem #1:
What if the first node hit any leaf nodes?



Idea #2:
For a given node check both children aabbs
Recurse into the first aabb that is hit
If a leaf node was hit then return
Otherwise recurse into the other side

What's wrong with this algorithm?

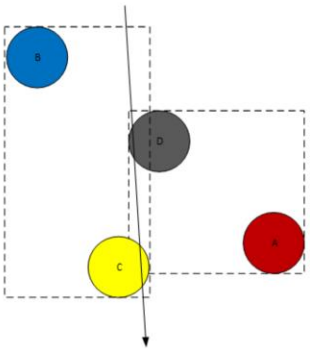
The most obvious error with the first algorithm is what happens if nothing is hit on this first side? The fix here is simply, just recurse down the other side then.

Unfortunately this doesn't catch everything…

Ray-Casting – First Impact

Problem #2:

What if the first hit leaf node isn't the t-first hit?



Since aabbs can overlap we can't assume that the first node we hit is will ever contain the first impact

Unfortunately, it's not too hard to make a scenario where the first leaf node that the algorithm would across is not actually the first hit node.

In the above example the left aabb will be recursed into first as it's the first hit. This will end up finding an intersection with the node C and return. However, the actual first hit node is node B.
How do we update the algorithm to deal with this?

Ray-Casting – First Impact

Idea #3:

For a given node check both children aabbs

Recurse into the first aabb that is hit

- If nothing was hit hit go down the other side

If a leaf was hit then check at what time.

If this time is before the starting t-value of the other aabb then you can safely skip it

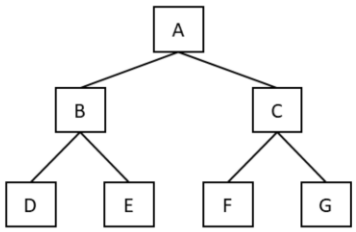
Otherwise go down the other aabb and keep the min of the leaf nodes

The final algorithm becomes quite messy.

The basic idea is that when we get a result from one side we have to see at what time it happened. If it happened before we could enter the other aabb then we can safely assume this is the first impact for the current sub-tree. Otherwise we don't if there's something else in the other aabb that could be hit first so we have to go down the other side and merge the results.

Pair Query

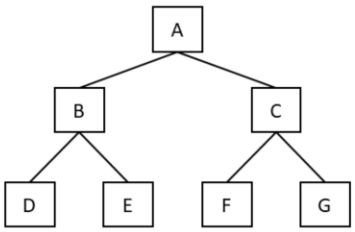
How do we determine what objects within the tree collide?



Now for the last kind of query we need to perform: pair (or self) queries. This query is most commonly performed by physics but can still be useful to other areas. A pair query is simply: what objects within this tree collide with other what other objects in the tree?

Pair Query

Method 1: Query each object against the tree



What problems does this have?

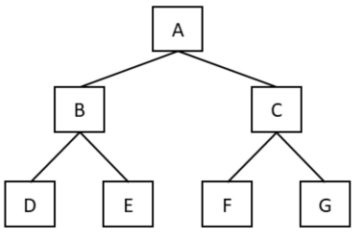
The first pair query method is to query each leaf child against the tree. That is, to find out what E collides with just send its aabb through the tree to get all pairs. Then do this for F, G, and so on.

This method obviously has some problems though…

Pair Query

Method 1 Problems:

- Each object will create a pair with itself
- Pairs will be created twice
- Wasteful when doing lots of queries



Can we do better?

The first problem with this method is that an object will form pairs with itself. We don't want to get self pairs as this is redundant and wasteful information (obviously E collides with E…).

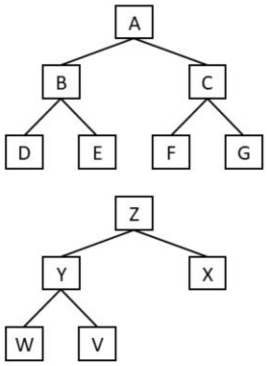
The second problem is that a pair will be created twice. Node D will find that it collides with E and E will find that it collides with D. This can easily be worked around by using a container that stores unique entries.

The biggest problem here is how wasteful this algorithm is. By definition, D and E will intersect nodes B and A. So we do 2 tests for something we already know. This means we'll spend a lot of time traversing top level nodes when ideally we'd like to use the information already in the tree to speed things up.

Tree Query

First look at how we would intersect two different trees

Traverse down both simultaneously



Bounding Volume Test Tree (BVTT)

How do we continue testing from here?

Before resolving how to perform pair queries we need to look at how we would intersect two independent trees together. To help I'll be using a visual aid known as a bounding volume test tree. This is a tree that records what pairs are tested as we traverse down the tree. Each node here will represent a collision test between two nodes.

To start colliding these two trees is easy, simply test the two roots against each other. If the roots don't collide then the entire trees can't collide. The question is how do we determine what remaining tests to perform?

Tree Query

Three cases to consider when testing two nodes:

1. Both are leaf nodes
2. One is leaf and one is internal
3. Both are internal

The only tricky case is if both are internal

*In all cases make sure to test the aabbs first

When we have two nodes to test for collision in our trees we have 3 possible cases to consider.
The first case is easy: both nodes are leaf nodes. The only logical thing to do with two leaf nodes is add them as result pairs.
The second case is also easy: we have a leaf and an internal node. We can only add a result pair of leaf nodes so we have to recurse. As the leaf node has no children the only choice is to split the internal node by testing both of its children against the leaf node.
This just leaves us with the final case: both nodes are internal nodes.

Make sure that at every point in the tree you test the two current nodes' aabbs for intersection. If the don't collide then none of these cases even matter.

Tree Query

Case 3: Both are internal

"Split" the nodes

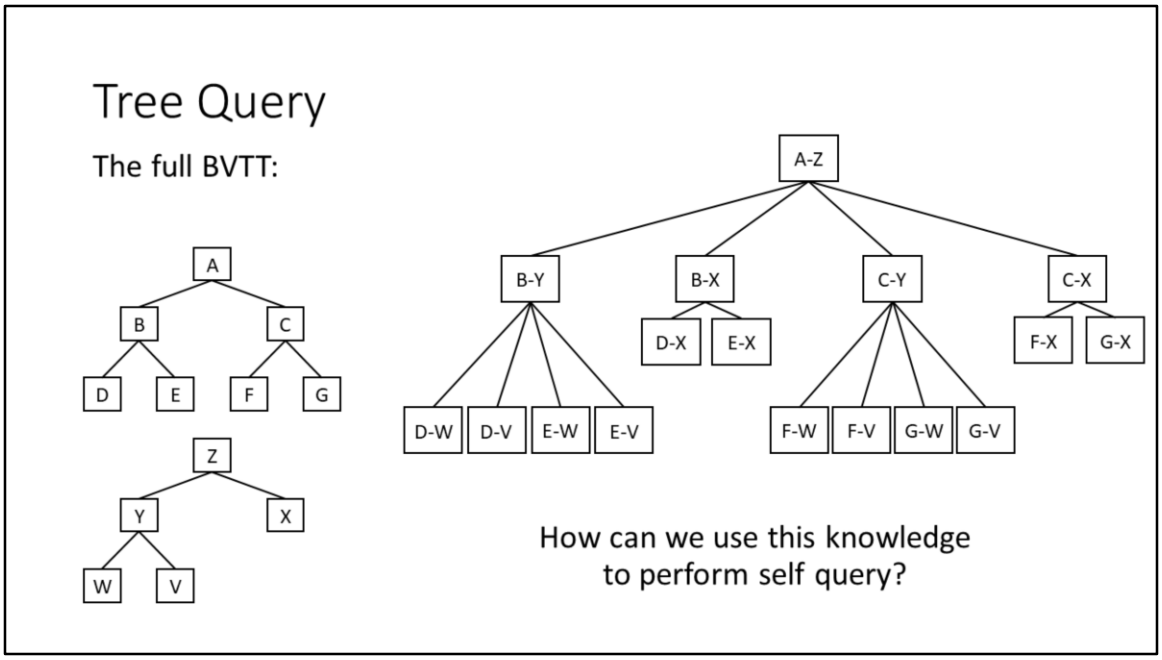
```
void TreeQuery(Node* nodeA, Node* nodeB)
{
  // Case 1: Both are leaf
  // Case 2: One internal, one leaf
  // Case 3: Both are internal, split the nodes
  else
    SplitNodes(nodeA, nodeB);
}
```

```
void SplitNodes(Node* nodeA, Node* nodeB)
{
  // Test all children pairs
  TreeQuery(nodeA->mLeft, nodeB->mLeft);
  TreeQuery(nodeA->mLeft, nodeB->mRight);
  TreeQuery(nodeA->mRight, nodeB->mLeft);
  TreeQuery(nodeA->mRight, nodeB->mRight);
}
```

*We'll revisit this SplitNodes function later

Case 3 requires us to somehow "split" the internal nodes. By split we must take at least one of the nodes and continue the recursion, but with its children.

For now we're going to assume that we do this by testing all 4 children permutations against each other. We'll revisit the split function later, just understand it conceptually for now.



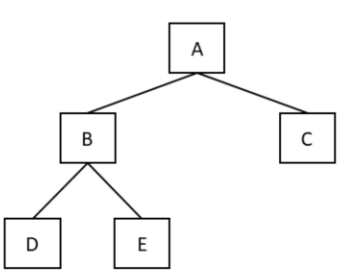
So given that we now know how to define all 3 cases of our pair query we can define what the full BVTT looks like.

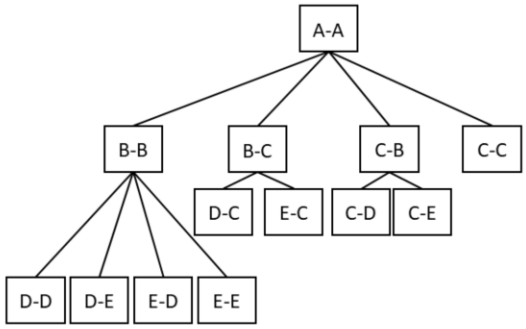
The important thing to notice here is that we tested all leaf nodes of both trees against each other with no duplicates and with a very small number of total nodes tests. To test all of these leaf nodes we performed 17 aabb vs. abbb tests. To perform this same test with the method of test each leaf node against the tree would've required 20 aabb vs. aabb tests.

The question now is how to use this test to efficiently perform a self query (pair query) test.

Pair Query – Using Tree Query

Attempt 1: Just query the tree against itself





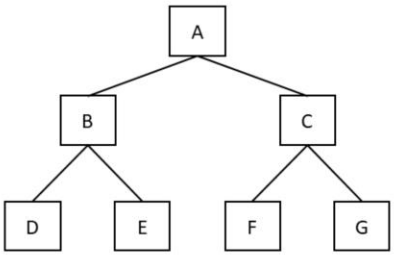
This is really bad!

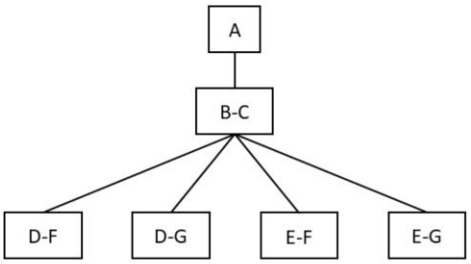
The most obvious first idea is to pretend that we have two trees and simply query the tree against itself, that is TreeQuery(A, A). From a quick BVTT it should be obvious that we produced a very bad tree. One could argue that we could simply ignore all node vs. itself cases and we'd get a much better tree, but we'd still be detecting a pair twice (B-C and C-B).

Luckily it's easy to make this significantly better!

Pair Query – Using Tree Query

Attempt 2: Query B and C as two different trees





Are there any problems?

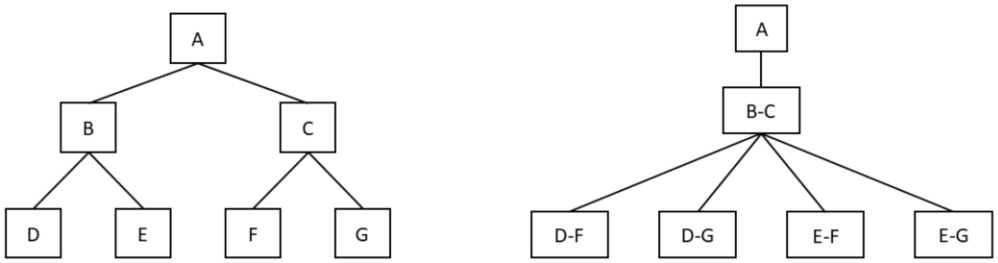
The easy way around this is to not ever test A against A as this will obviously lead into problems. Instead we can take the first two sub-trees and query them against each other.

We can write out the BVTT again and see that this tree is much better looking. This leads to the question, are there any problems here?

Pair Query – Using Tree Query

No node duplicates

Are nodes missing? We should have 6 pairs…



We're missing D-E and F-G. How?

Well there's no duplicate pairs in this method, but are there missing pairs. Remember that we want to efficiently get all pairs for testing (assuming all aabbs hit) so we should have every leaf node paired up together. If we count this out we should have a total of 6 pairs instead of 4. We're missing the pairs: D-E and F-G.

How does this happen

Pair Query

Attempt 2: Add missing pair checks

```
void SplitNodes(Node* nodeA, Node* nodeB)
{
  // Test all children pairs
  TreeQuery(nodeA->mLeft, nodeB->mLeft);
  TreeQuery(nodeA->mLeft, nodeB->mRight);
  TreeQuery(nodeA->mRight, nodeB->mLeft);
  TreeQuery(nodeA->mRight, nodeB->mRight);
}
```



```
void SplitNodes(Node* nodeA, Node* nodeB)
{
  // Test all children pairs
  TreeQuery(nodeA->mLeft, nodeB->mLeft);
  TreeQuery(nodeA->mLeft, nodeB->mRight);
  TreeQuery(nodeA->mRight, nodeB->mLeft);
  TreeQuery(nodeA->mRight, nodeB->mRight);
  // Add missing pairs
  TreeQuery(nodeA->mLeft, nodeA->mRight);
  TreeQuery(nodeB->mLeft, nodeB->mRight);
}
```

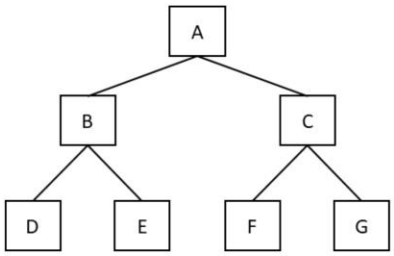
Does this catch everything?

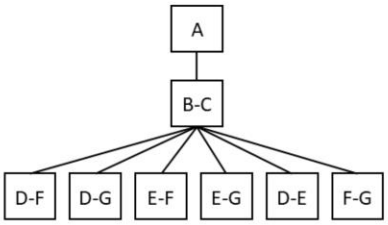
If we look at the split nodes function it should be obvious. We check all of the permutations of nodeA with nodeB but we don't check the children of nodeA with themselves. So we can simply update split nodes and get our missing pairs and wrap this all up!

Just to be sure though, maybe we should check the pairs again.

Pair Query – Using Tree Query

No duplicates, no missing pairs!





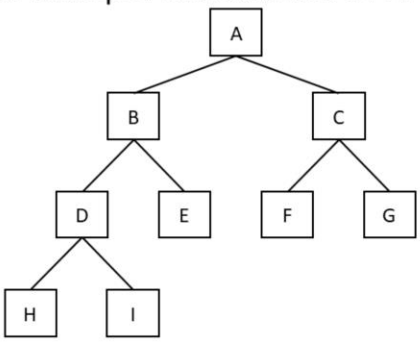
Are there any problems left?

Well we can write out the BVTT again and see that this tree looks perfect. Once again the question is: are there any remaining problems.

From what we can see here so far there aren't, but maybe we just have the wrong example…

Pair Query

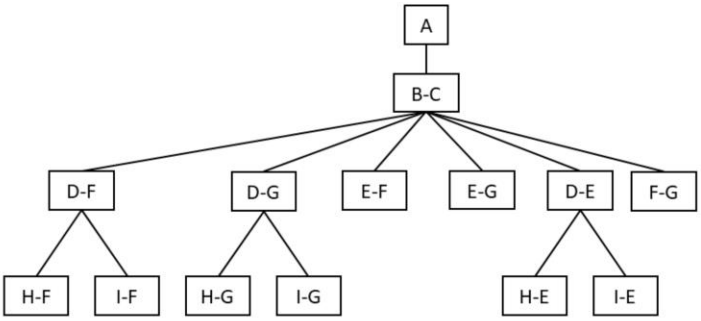
Try a slightly bigger example: Evaluate this BVTT



Maybe if we try a bigger tree we could see the problem, but how much bigger? Luckily it doesn't take much more, however the BVTT grows quite huge so I've put it on another slide.

Pair Query

BVTT:



Is there anything wrong?

It might be hard to see at first, but there is something wrong with this tree.

Pair Query

Given the leaf nodes:

H	I	E	F	G

We found the pairs:

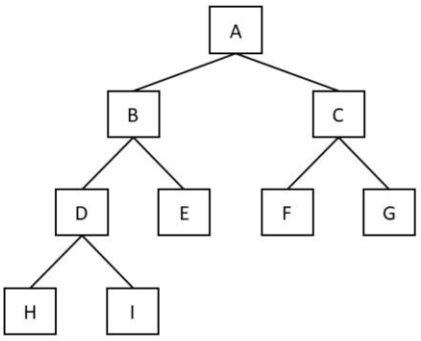
H-F	I-F	H-G	I-G	E-F	E-G	H-E	I-E	F-G

What's missing?

If we inspect a bit further we can see that we had 5 leaf node children in the original tree. We also ended up with 9 leaf node pair tests. Does this match up?

Pair Query

We never tested H-I



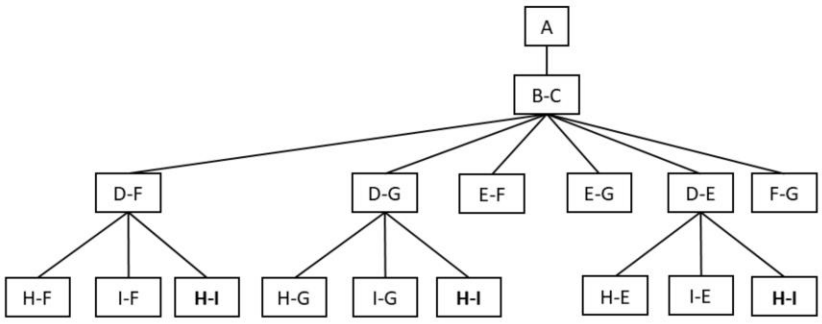
How can we fix this?

Modify case 2 to check the two children?

We're missing one pair that we needed, test H vs. I. The quest is how do we fix this now? One way would be to modify case 2 (internal vs. leaf) to check the two children of the internal node against each other. What happens if we do this?

Pair Query

BVTT: Modified case 2



It might be hard to see at first, but we added 1 pair here 3 times. Adding this pair more than once might not seem like too big of an issue as we can just add a hashmap to remove redundant pairs, right? Unfortunately these pairs we generate will be at the top of the tree and cause many extra aabb checks. If implemented this way the query time can be several times larger than if we had just done the leaf node queries. This is clearly not a viable option, and it only grows worse with larger trees.

If you don't believe me then try the large level in the framework with this method, it'll perform worse than an n^2 spatial partition…

Pair Query

Solution 1: Add flags for each pair

- Keep a flag for if a child has already been checked

Problems:

- You have to clear the flag
- Not thread safe

One possible solution is to add a flag to keep track of whether a node's 2 children have been checked against each other. Keeping an actual flag is a bit cumbersome as we have to clear it later. We could instead use a timestamp but this not only can break over time as the tree updates (our timestamp can eventually wrap around and if we haven't checked a pair in a long time this will break) but it is also not thread safe for performing multiple queries at the same time. That being said, this is a reasonable first approach, but can we do better?

Pair Query

Split the recursive function into 2 functions and revert SplitNodes

```
void SelfQuery(Node* node)
{
  if(node->IsLeaf())
    return;

  // Check the two children against each other
  SelfQuery(node->mLeft, node->mRight);
  // Recurse on the left and the right tree
  SelfQuery(node->mLeft);
  SelfQuery(node->mRight);
}
```

```
void SelfQuery(Node* nodeA, Node* nodeB)
{
  // Case 1: Both are leaf
  // Case 2: One internal, one leaf
  // Case 3: Both are internal, split the nodes
  else
    SplitNodes(nodeA, nodeB);
}
```

```
void SplitNodes(Node* nodeA, Node* nodeB)
{
  // Test all children pairs
  SelfQuery(nodeA->mLeft, nodeB->mLeft);
  SelfQuery(nodeA->mLeft, nodeB->mRight);
  SelfQuery(nodeA->mRight, nodeB->mLeft);
  SelfQuery(nodeA->mRight, nodeB->mRight);
}
```

It turns out we can split our recursive function into 2 recursive functions. In so doing we can write the algorithm requiring no flags, getting all pairs, and producing no duplicate pairs. Basically we have 1 function that is our original tree query function that would miss pairs by using the original SplitNodes function. Then we create another function that recurses a sub tree by checking the 2 children as trees and then recursing on each of those sub-trees.

The only problem with this algorithm is that it's harder to turn from a recursive function to a stack function, although not by much.

This is what you should implement for your assignment.

Split Nodes

There's many ways to split nodes

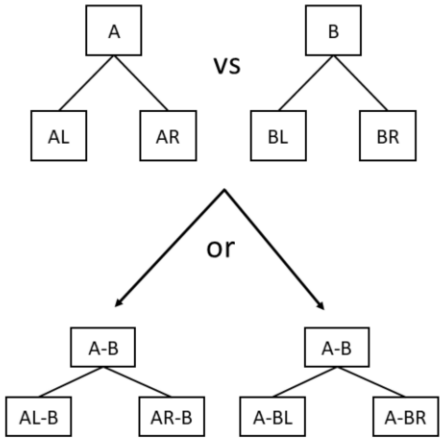
This just leaves the final talk about the SplitNodes function.

At a high level there's 3 different ways we can choose to split nodes.

Split Nodes

There's many ways to split nodes

- Always split one side



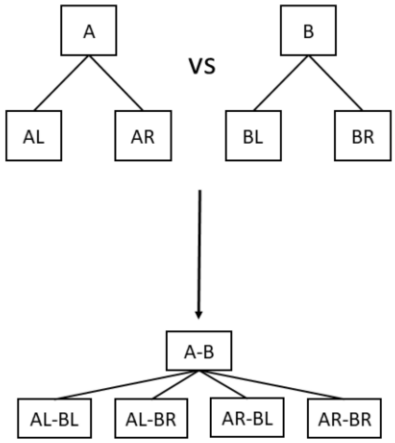
The simplest method would be to always split one side. This will drill down into one tree as quickly as possible, but is this really a good idea? If we had special knowledge about a tree then maybe we could know that we always want to drill deeper into one first but that's unlikely.

For example. Imagine we have a model of DigiPen and a model of a bird. The bird's tree takes up much less space than DigiPen so it would make more sense to drill first into DigiPen's tree, but at this point we get very little saving when it comes to any piece of DigiPen against the bird. It's likely that the bird won't hit much so this shouldn't be too bad though (if we had this knowledge).

Split Nodes

There's many ways to split nodes

- Always split one side
- Always split both



The second option is the one we've already seen: to split both. This one will often times perform better than the exclusive splitting of one side, but it does have times where it won't.

It should seem clear that neither of these choices will always be the best answer. We clearly need some conditional logic!

Split Nodes

There's many ways to split nodes

- Always split one side
- Always split both
- Conditionally split one side (heuristic)

Finally we can implement some kind of heuristic to sometimes split one side or sometimes split the other. This has the benefit of producing good results regardless of what kinds of input trees we have.

Split Nodes - Heuristics

What heuristics can we use?

- Largest surface area
- Largest volume
- etc…

For your assignment split the node with the largest volume!

Now we can come up with any heuristic we desire. The two most common ones are to split a node based upon which has the largest surface area or which has the largest volume

From the small amount of testing I've done, it doesn't seem to matter too much which one you use. That being said for your assignment you will split the node with the largest volume.

Questions?