# Sabanci University

## Faculty of engineering and Natural Sciences

## CS 300 Data Structures

Assigned: **Nov 19, 2020** Due: **Nov 30, 2020 @ 11:55pm**

**PLEASE NOTE**

**SOLUTIONS HAVE TO BE YOUR OWN. NO COLLABORATION OR COOPERATION AMONG STUDENTS IS PERMITTED.**

**10% PENALTY WILL BE INCURRED FOR EACH DAY OF OVERTIME. SUBMISSIONS THAT ARE LATE MORE THAN 3 DAYS WILL NOT GET ANY CREDITS.**

**SUBMISSIONS WILL BE MADE TO THE SUCOURSE SERVER. NO OTHER METHOD OF SUBMISSION WILL BE ACCEPTED**.

# Introduction

Game development is a very challenging process involving many different problems. This includes everything from rendering graphics, carrying out game mechanics, responding to user input, and creating the AI. In this assignment, we will explore a very common problem in game development, and physics engines in general; the problem of collision detection. Let's look at the following screenshot:


Figure 1: a screenshot from the game Super Smash Bros. Ultimate, the latest installment from (objectively) the greatest fighting game series of all time.

We can see there are many characters on the screen and many projectiles flying around. The question is, how does the game figure out that two objects have touched each other? Resolving the problem of two objects “touching” in a physical space is called collision detection. One way to handle this problem is to split each object into smaller “member” shapes that are easier to check than weird (very curvy) shapes. These shapes are often referred to as hitboxes. For example, the previous frame can be seen by the physics engine as the following collection of members:

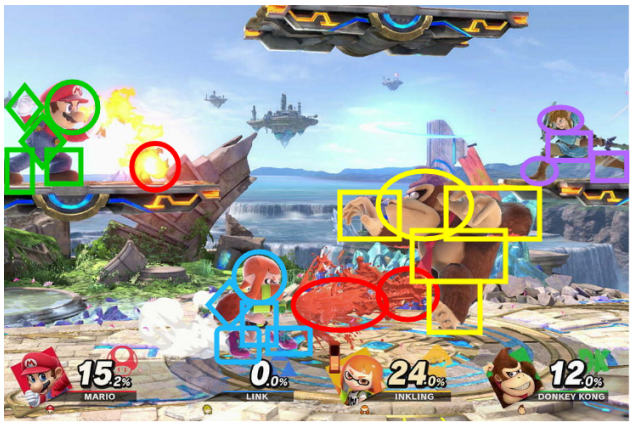


Figure 2: the member shapes (hitboxes) of the objects in the game. Notice the variety of shapes and how many there are.

Now, when we wish to check whether two objects are touching, we need to compare the locations of the objects and see if their shapes are colliding. However, looking at this screenshot, we can see that this process can be costly. In the worst case, for each shape shown, we need to check *every other object* on the screen to see if a collision has happened. This can become costly fast. But there is a way to make this process less costly!

A suitable solution for this problem is to use **Bounding Volume Hierarchies (BVH)** to represent objects. Basically, with this approach, we represent each object as a *tree* of boxes that hierarchically cover the object, with the root node covering the entire object, and children covering smaller and smaller parts. The leaves of this tree are the original members of the object (its hitboxes), while non-leaf nodes are “branch” nodes whose job is to contain their children. Take for example the problem of collision detection demonstrated in Figure 3. We wish to find the objects that the blue projectile (a.k.a hadouken) is colliding with at this exact instance.


Figure 3: When using BVH, each object will be represented using a tree whose root node will cover the entire object. This means that there will be three trees, one for each object.

Each one of these objects (Ryu, Honda, and the hadouken) is an object. The characters are BVH trees (represented by a hierarchy of boxes) while the blue projectile is a single box. Figure 4 shows the objects on the screen. The boxes we see around the characters represent the root node of their BVHTrees.

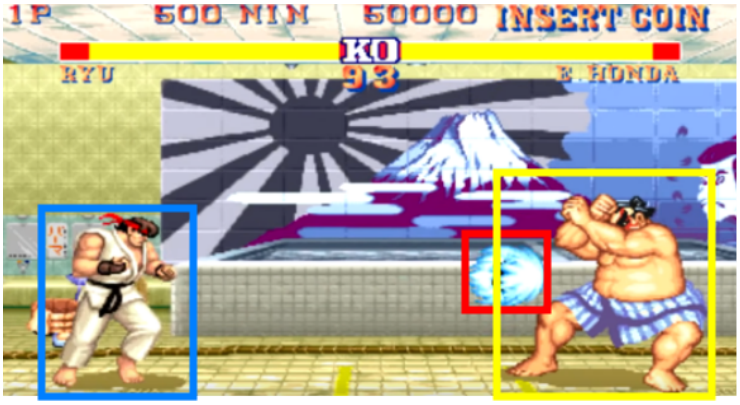


Figure 4: we wish to check whether the blue projectile (inside the red box) collides with any of the objects on screen. We find that the projectile collides with the root node of the rightmost BVHTree object.

Now, to see which objects the blue projectile (inside the red box) is colliding with, we only need to check two other boxes (the boxes at the root of the two characters' trees). When we do this check, we find that the projectile is colliding with the root of the tree of the character to the right only. Now, we begin searching for the exact collision point (if there is one) within the tree of that character. We do so by iteratively exploring the tree starting at the root. At each level, if we find that the projectile collides with a box, we will explore its children. If the box doesn't have children, it means that it's a leaf, and that it's one of the object's members.

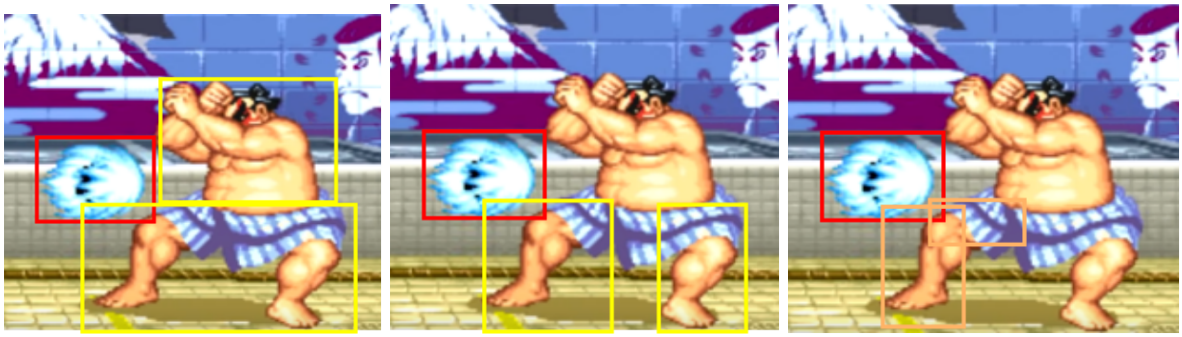

Figure 5: [left] after figuring out that the projectile collides with the root node of this character (Figure 4), we explore the children nodes of the root (branch nodes). We find that the projectile collides with the lower box so we explore its children. [center] we find that the projectile collides with the leftmost branch node, so we explore its children. [Right] We find that the projectile collides with two nodes. Both of these nodes are leaves, which means that the projectile indeed collides with the character represented by these members (hitboxes).

As shown in Figure 5, we iteratively explore the tree by checking all the boxes that collide with the projectile. During our exploration, when we find a collision with a leaf node (without any children), this indicates that a collision between the projectile and the character has happened at that member (hitbox).

This method of detecting collisions has many different algorithms, specializations, and data structures. In this assignment, you will implement a version of this algorithm that will represent boxes using Axis-Aligned Bounded Boxes (AABB). An AABB is a right angle rectangle (i.e., it does not include curves or weird angles). In addition, in this algorithm, the BVH structure you build will be a binary tree, i.e., each node will have no more than two children. To represent the AABB, we will use two points on the box, the bottom left corner located at (minX, minY), and the upper right corner located at (maxX, maxY). You will use the following struct to represent AABBs (note: a version of this struct with all the function implementations is attached with the document):

```
struct AABB{
      // The coordinates that define the AABB
      // (minX, minY) is the point that defines the bottom left corner
      // (maxX, maxY) the point that defines the upper right corner
      //  Example: A = (minX, minY), B = (maxX, maxY)
      //  +---B
      //  |   |
      //  A---+
      int minX, minY, maxX, maxY;
      // Construct the AABB
      AABB(int _minX, int _minY, int _maxX, int _maxY);
      // get the area of the current AABB
```

```
        int getArea() const;
        // Create an AABB that contains both the calling object
        // and rhs
        AABB operator+(const AABB& rhs) const;
        // Check if the calling object and the AABB rhs are identical
        bool operator!=(const AABB& rhs);
        // Check if the calling object and the AABB rhs collide
        bool collide(const AABB& rhs) const;
        // Print the information of this AABB to the stream "out"
        void printAABB(std::ostream& out);
        // Get the minimum value between x and y
        static int getMin(int x, int y);
        // Get the maximum value between x and y
        static int getMax(int x, int y);
        // Return the area of the AABB that will cover (or contain)
        // the AABBs lhs and rhs
        static int unionArea(const AABB& lhs, const AABB& rhs);
};
```

You will write the algorithm to construct a BVH for an object, which we will call the “agent”. Then, you will (a) check for collisions between the BVH and external projectiles and (b) you will modify the sizes and locations of the members within the agent. In the following sections, we will detail exactly how these operations are to be carried out.

It is important to note during the execution of the program, you will build a *single* BVH corresponding to a single agent, and you will check if projectiles (each projectile being represented by a single AABB) collide with this agent. In other words, you will be doing the algorithm demonstrated in Figures 4 and 5.

# Building the BVH

The BVH class you will implement will have the following structure:

```
class BVHTree {
private:
    BVHTreeNode *root;
    std::unordered_map<std::string, BVHTreeNode *> map;
public:
    BVHTree();
    ~BVHTree();
    void printNode(std::ostream &out, BVHTreeNode *, int level);
    void addBVHMember(AABB objectArea, std::string name);
```

```
    void moveBVHMember(std::string name, AABB newLocation);
    void removeBVHMember(std::string name);
    std::vector<std::string> getCollidingObjects(AABB object);
    friend std::ostream &operator<<(std::ostream &out, BVHTree &tree);
};
```

It will contain a pointer to the root of the tree, as well as an unordered map (hashtable) which will map names of components to pointers at these components in the tree. You must implement the constructor, destructor and the functions addBVHMember, moveBVHMember, removeBVHMember, and getCollidingObjects. You don't need to implement the operator<< and printNode functions, as they are provided in the file "BVHTree.cpp" that's given with the homework document in the src.zip file.

The tree nodes will have the following structure:

```
struct BVHTreeNode{
      BVHTreeNode *parent, *leftChild, *rightChild;
      AABB area;
      std::string name;
      bool isLeaf;
      BVHTreeNode(BVH _area, std::string _name, bool _isLeaf);
};
```

The parent pointer will point to the parent of this node, and the leftChild and rightChild will point to its left and right child (if any), respectively. The string name will contain the name of this node, and the boolean isLeaf will indicate whether or not this node represents a member (hitbox) of the agent or an intermediate node (i.e., a branch).

As a hint, we want to draw your attention to two important properties (invariants) of the tree:

1. The members of an object (hitboxes) are *always* the leaves of the tree. Any node that isn't a leaf must be a branch node - a node that doesn't represent a member.
2. Leaf nodes (i.e. member/hitbox nodes) have no children.
3. Branch nodes *must always have exactly two children.*

After finishing any operation during the lifetime of your tree, you must ensure that these three invariants hold. If any of them doesn't then you've done something wrong.

## addBVHMember(AABB objectArea, string name)

This function will take an AABB `objectArea` and a string `name` for the member that is added. You must create a new `BVHTreeNode` with the area and name given. In addition, set the node to be a leaf, which indicates that it represents a member, not a branch. We will call this

node `newNode`. When adding this node to the tree, there are three possible cases you will face:

1. The tree is empty
2. There is exactly one node in the tree
3. There is more than one node in the tree

## Tree is empty

If the tree is empty, set the root of the tree to be the newly created node. An example of this is shown in Figure 6.

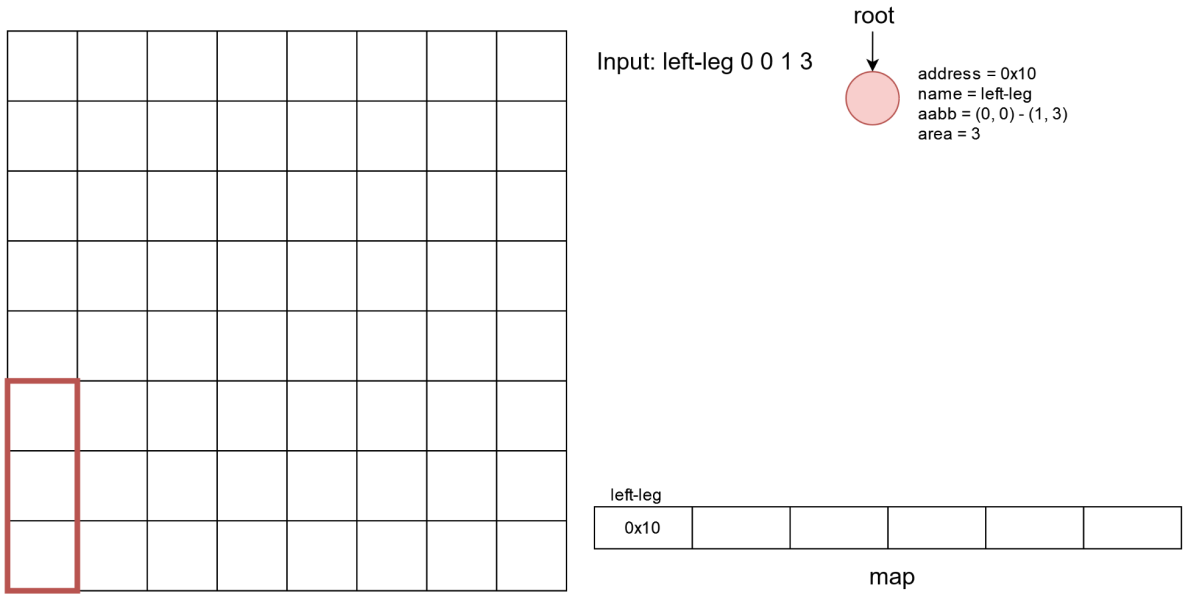


Figure 6: This is the result of adding the very first member to the BVH. The components name and its minimum point (bottom left corner) and maximum point (top right corner) are shown in the input. The figure on the left is only for demonstration purposes and isn't part of the code. When the node is added, it becomes the root of the tree, and the hashmap "map" will map its name to its address.

## There is exactly one node in the tree

Let's call the current node at the root `oldRoot`. You must create a new branch node (non-leaf), which you will set as the root of the tree. In addition, you will set the left child of the branch node as `newNode` and the right child as `oldRoot`. And, importantly, you will set the area of the branch node to be as big as *both* children. In other words, the area of the branch must cover its childrens areas. This is an important insight: **anytime a node is added as a child of another node, the parent node as well as all of the parent's ancestors must adjust their area to fit the new child node.**

This step is demonstrated in Figure 7. It's important to note here the area of the root of the agent has increased. This is not very desirable, as it means there will be more empty space in the AABB of the agent. However, this is a price we have to pay for better performance.

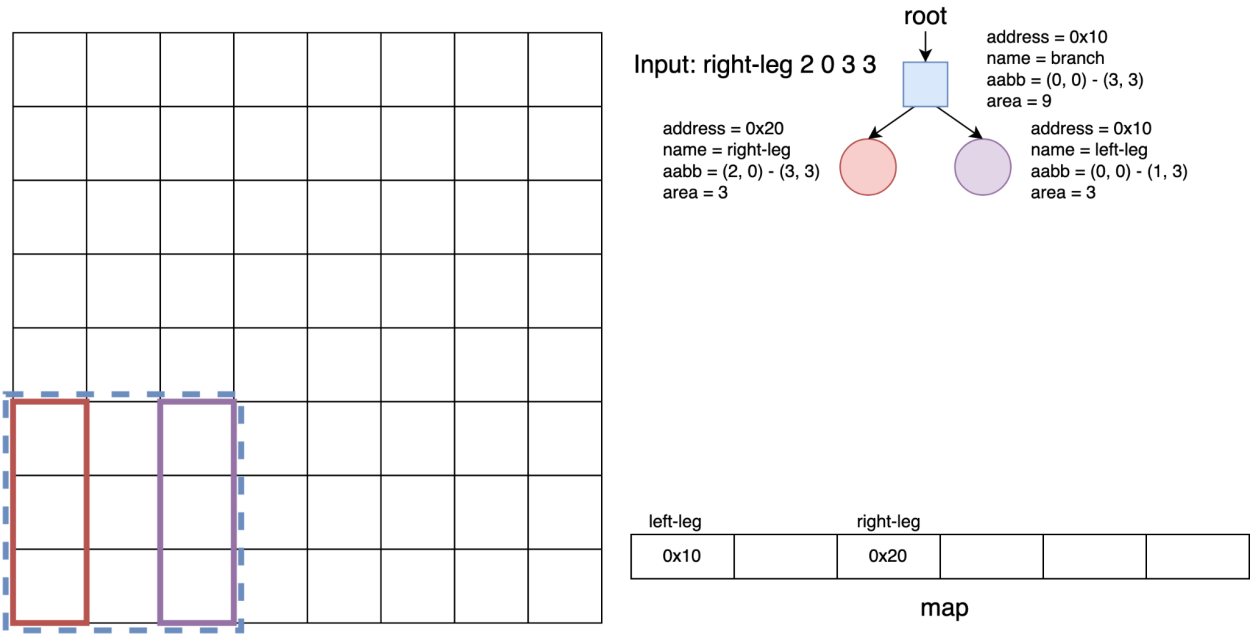


Figure 7: When adding a second member to the BVH, we must create a branch node that will point at the old and new nodes. Notice that the root of the BVH has the size of both nodes combined.

## There is more than one node in the tree

In this case, we must find a location for this node somewhere in the tree. Recall from the invariants mentioned earlier that, in our tree, member (hitbox) nodes cannot have children and branch nodes always have two children. So, if we wanted to add a new node to an existing tree, we cannot add it as a child of an existing branch node since all branch nodes already have two children, and we cannot make it a child of a leaf node since it will no longer be a leaf. So, what we *must* do, is:

1. Pick a leaf node in the tree, call it existingLeaf
2. Create a new branch node that will take its position
3. Set `newNode` as the left child of the new branch node and `existingLeaf` as its right child.

Notice that this operation will maintain our tree invariants. The leaf nodes `newNode` and `existingLeaf` are still leaves (have no children,) and the new branch node has exactly two children.

An important question arises at this point; how do we pick `existingLeaf`? Unlike binary search trees, there is no ordering property between sibling nodes. The only condition we must maintain is that this node must be a leaf node. So then how should we proceed? Where do we place the node? Turns out there are many different algorithms for finding an appropriate location for new nodes in the tree, and each comes with its own benefits. For this assignment, we will attempt to place new nodes in locations that will minimize the increase in other nodes' areas.

We do so by iterating down the tree starting at `root` until we reach a leaf. At each branch node, we will go in the direction of the child whose area would increase the least if `newNode` becomes one of its children. More precisely, given that we wish to add `newNode` to the tree, and given that we are at the branch node `branchNode`, we will calculate the following two values

```
int increaseInRightTreeSize = AABB::unionArea(newNode->aabb,
branchNode->rightChild->aabb) - branchNode->rightChild->aabb.getArea();
int increaseInLeftTreeSize = AABB::unionArea(newNode->aabb,
branchNode->leftChild->aabb) - branchNode->leftChild->aabb.getArea();
```

And if `increaseInRightTreeSize < increaseInLeftTreeSize`, we will go to the right child. Otherwise, we will go to the left child.

At this point, we'd like to remind you of the following rule: **anytime a node is added as a child of another node, the parent node as well as all of the parent's ancestors must adjust their area to fit the new child node.** So, once a new node is added, make sure that all of its parents adjust their areas to contain it.

An example of this case is shown in Figure 8.

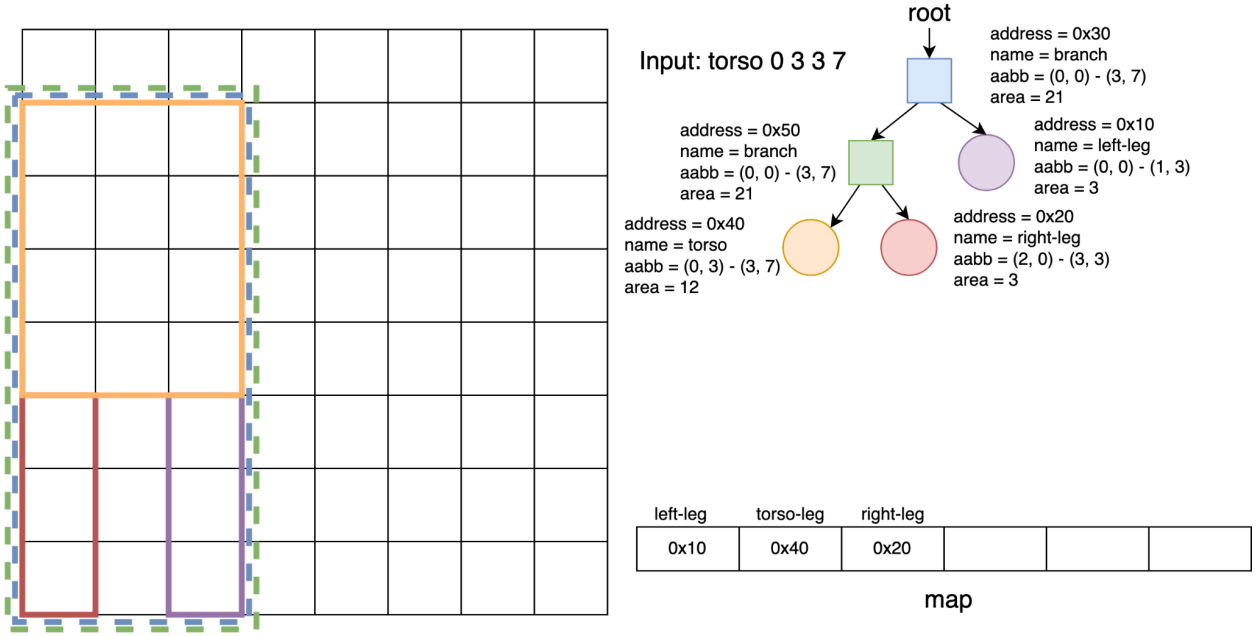


Figure 8: A new branch node is created to point at the old leaf and the newly added one. Notice that the increase in size is carried from the green branch node to all of its ancestors. This is because a parent of a node must cover its entire AABB.

## removeBVHMember(string name)

When we remove a node from the tree, we must make sure to maintain the invariants of the data structure: leaves have no children, branches have exactly two children, members are leaves and branches aren't. When we want to remove a member node, we will leave its parent branch node with only a single child, which will break the invariant. For this reason, when we remove a node, call it `toRemove`, we will replace the branch node that is the parent of `toRemove` by the *sibling* of `toRemove`. This way, the grand parent of `toRemove` will have two children and the leaves will have no children.

Don't forget to remove the pointer of this node from the hashmap!

## moveBVHMember(string name, AABB newLocation)

This function will change the location and dimension of a member node. When this function is called, there are two possibilities:

1. The new location is covered by the parent of the node, in which case you will only change the location (AABB) of the node itself (not its parents).
2. The new location of the node is *not* covered by its parent. In this case, you will remove the node and reinsert it in the tree.

## getCollidingObjects(AABB location)

This function takes an AABB object and will store the names of all the members that collide with this object. You must start a search from the root and go to its children. At each node, if the node's AABB overlaps with location (you can check this using the overlap() function in the AABB struct), then:

1. If it's a leaf, you will add it to the vector of names of collided members
2. If it's a branch, you will check its children.

Important: You don't need to worry about the order of finding the names. As long as you find the correct names then you are safe.

# Program flow

For this assignment, you are given a big portion of the code, and you must complete the missing portion. More specifically, you are given full implementations of the the `BVHTreeNode` struct, the `AABB` struct, and the main function. In addition you are given the header file of the `BVHTree` class, the implementation of two functions from the `BVHTree` class.

You are not allowed to change any of the existing functions given in the code. However, you may (in fact, you are encouraged to) add *additional* helper functions to the classes if that will help you write the required functions. To summarize, you only need to implement the functions of the `BVHTree` class.

## Main program

The main program will read two text files, “agent.txt” and “actions.txt”. “agent.txt” will contain the member data of the agent, and the “actions.txt” file contains the actions that must be done on the BVH. The main program will use the functions that you will implement to build the BVH that satisfies “agent.txt”. Afterwards, “actions.txt” will be read and its actions will be performed on the BVH that you created. The actions that will be performed are the following:

## Collision detection with single AABB

Syntax:
`c minX minY maxX maxY`

You must find all the members of the constructed BVH that collide with the AABB object located between the points (minX, minY) and (maxX, maxY).

## Moving a member inside the BVH to a new location

Syntax:
`m member_name minX minY maxX maxY`

Move the member with the name “member_name” to the new location given by the AABB object located between the points (minX, minY) and (maxX, maxY).

## Print out the tree

Syntax:
`p`

Print the current BVH structure. This function has already been implemented for you.

# Frequently asked questions

1. Does my tree structure *have* to match your tree structure? What if my right child and left child are reversed? Or are my children in the wrong order?

Your tree structure *must* match the grader's tree structure. I.e., there is a single correct tree structure for any question. If you follow the rules listed in the document then you will always arrive at the same tree.

2. What if the tree structure is different, but my collision detection function returns correct results?
   Your tree structure must match the correct structure even if your collision detection function produces the correct values.
3. Are we going to check the collision between two BVH trees?
   No. You will only check for collisions between a single AABB object and a tree.
4. What should be the value of the "name" member in the AABB struct of branch nodes?
   You can use "branch" as the name (as shown in Figures 6-8 but it shouldn't affect the correctness of your code.

# Sample Run

Given the input text files:

agent.txt

```
left-leg 0 0 1 3
right-leg 2 0 3 3
torso 0 3 3 7
head 1 7 3 8
```

actions.txt

```
c 1 2 2 3
c 10 10 11 11
c 0 3 1 8
m left-leg 1 0 2 3
c 0 3 1 8
p
```

We will get the following output (bolded text is user input):

```
Projectile (1, 2), (2, 3)
Collides with: left-leg, right-leg, torso
Projectile (10, 10), (11, 11)
Collides with:
Projectile (0, 3), (1, 8)
Collides with: head, left-leg, torso
Moved the left-leg to the location (1, 0), (2, 3)
Projectile (0, 3), (1, 8)
Collides with: head, left-leg, torso
```

```
Current tree:
+ branch || min = (0, 0), max = (3, 8), Area = 24
  - R - leaf: left-leg || min = (1, 0), max = (2, 3), Area = 3
  + branch || min = (0, 0), max = (3, 8), Area = 24
      - R - leaf: right-leg || min = (2, 0), max = (3, 3), Area = 3
      + branch || min = (0, 3), max = (3, 8), Area = 15
      - R - leaf: torso || min = (0, 3), max = (3, 7), Area = 12
      - L - leaf: head || min = (1, 7), max = (3, 8), Area = 2
```

# Submission

Your code should be submitted to SUCourse+ at the deadline given on the first page. You should follow the following steps:

- Name the folder containing your source files as *XXXX-NameLastname* where *XXXX* is your student number. Make sure you do NOT use any Turkish characters in the folder name. You should remove any folders containing executables (Debug or Release), since they take up too much space.
- Compress your folder to a compressed file named, for example, *5432-AliMehmetoglu.zip*. After you compress, please make sure it uncompresses properly and reproduces your folder exactly.
- You then submit this compressed file in accordance with the deadlines above. Your homework will be graded in the following way:
    - If your program does not construct and query BVH trees as described in the attached document, you will get 0 points. This will be the case even if your program works correctly otherwise.
    - We will run about 10 tests on your homework. Each correct test will earn you 10 points. Note that your outputs should be in the exact same format we described above.

Good luck