# CS5346: Fall 2018

# Project 2: Kalah

Submitted by: Gentry Atkinson

Teammate: Vishal Kumar

# Table of Contents

# 1. Introduction:

Mancala is one of the oldest known games. It and its many variations are still played in many parts of the world. It is conceptually and materially very simply. However, the strategy can still be very challenging. This makes it very well suited to the exploration of of two-player searches implementing intelligent evaluation functions.

Kalah is a variation of Mancala which was developed in the mid-20th century for American audiences. The Kalah game board gives each player a certain number of holes (or houses) and starts with a certain number of stones (or seeds) in each hole. A standard notation was introduced to describe versions of Kalah with varying numbers of holes and stones. Generally Kalah(m, n) describes a game being played with m holes per player and n stones per hole. This project has chosen to focus on Kalah(6, 6), meaning that all games described in this project are being played with with 2 players, 12 holes, 72 stones, and 2 kalahs (or stores). The use of each of these will be described in the following section.

The easy rule set and surprisingly difficult strategy of Kalah make it ideal as a vehicle with which to study intelligent 2-player searches. Human players can quickly develop an intuitive sense for ideal board states but computers do not share the benefits of this intuition. The understanding of the human players must be distilled into tight, manageable rule sets which a computer can then apply. There are several conditions which can easily be seen as being desirable but it is not always easy to say which of these desirable conditions is more desirable than its mates. This is why an intelligent computer or human must look past the immediate state of the board and choose a strategy which benefits them several moves into the future.

Kalah(6,6) has been solved by computer systems. This means that a player who has the first move can guarantee a when if they play an optimal strategy. However this optimal strategy depends on the existence of a full game database. Any turn of Kalah can have up to 6 legal moves, turns can repeat, and the game can last for more than a dozen turns. This means that the full tree generated by Kalah can be extremely large. In order to avoid the large expenditures of space and time which are required by full game databases, this project will attempt to use smart searching algorithms which can limit their search depth to a manageable level while still producing a near optimal result. In order to allow the system to judge the optimality of an unfinished game, a pair of evaluation functions have also been developed.

The purpose of this project is to develop and compare two methods of evaluating a board's state and two methods of search a tree of board states to back-propagate the values of the

leaf nodes in the tree. These search algorithms also each include a cut-off which determines when a branch is no longer worth searching in order to optimize the execution time of this search.

Each author of this project has developed and contributed on evaluation algorithm. These algorithms are simply referred to by the author's first names in the code base for simplicity's sake. The performance of these algorithms is compared by playing a series of games with each algorithm choosing the moves for one player. In order to account for all factors which might affect the algorithms' performances these games are played over a variety of tree depths and with each of the two search algorithms.

Similarly, two optimal search algorithms have been chosen from the literature: Alpha Beta Search and MinMax AB. These algorithms have been implemented in the code base of the project and their performances will evaluated against one another in a similar fashion to the two evaluation algorithms. While the evaluation algorithms were compared with each search algorithm in place to assure fairness, the search algorithms will be compared with each evaluation algorithm in place to insure that they also each have a fair chance.

After the test cases have been executed, this project will collect and present the results. Some ad hoc conclusions can be drawn about the performance of each algorithm from analysis alone but in the end data speaks loudest. The results will be tabulated and presented so as to present whatever clear distinctions in the code's performances can be made.

## 1.1 Rules of Kalah:

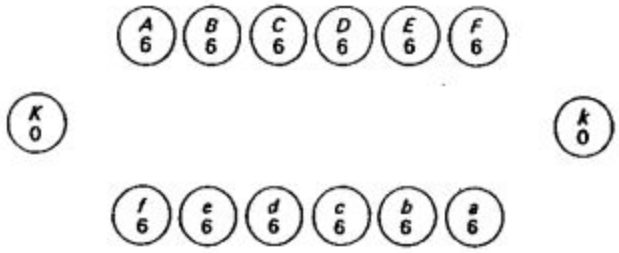


**Figure 2-8** Starting position in kalah. There are six stones in each hole and none in each kalah.

1. Each player begins the games with 6 holes, 1 kalah, and 6 stones in each hole. Each player owns the kalah that is to his or her right and the six holes closest to him or her.
2. The objective of the game is to have more than have of the stones placed in the kalah owned by a player.
3. If all of the holes of one player become empty, all of the stones in his or her opponent's holes are are placed in the first player's kalah and the game ends.
4. A player makes a move by picking up all of the stones in one of his or her own holes. These stones are then distributed one at a time into the holes around the board. A player places stones into his or her own kalah while distributing in this fashion but not into his or her opponent's.
5. If the last stone of a player's move is into the kalah, that player goes again.
6. If the last stone of a player's move is placed into an empty hole belonging to that player, all of the stones in the adjacent hole (across the board) are place into that player's kalah along with the one stone from the player's side.

# 2. Contributions:

Gentry Atkinson:

- Structure and object design
- Evaluation algorithm
- Code consolidation

Vishal Kumaar:

- Search algorithm implementation
- Code review
- Evaluation algorithm

This project would like to acknowledge John McCarthy for his development of the simplified rules for the game of Kalah as used in this project.

# 3. Analysis of the Problem:

Adversarial searches are most often considered with regard to game playing but they are actually much more broadly applicable. Any time that some actor must choose an optimal strategy in conditions that are not wholly determined by his or her actions then it is best to assume that those outside conditions are being determined by some adversary who will inevitably choose the "worst" possible conditions for the actor. This is the origin of min-max searching. An actor always chooses the maximal path while assuming that some adversary will "choose" the minimal path. Some examples of real world problems that benefit from min-max searching outside of gameplay: logistics, code optimization, personnel management, and epidemiology.

But min-max searching is not a simple problem so it is natural that many algorithms have been generated in order to accomplish this task. Two of the most well known are MinMaxAB and AlphaBeta Search. It is possible that each of these algorithms have an ideal application at which they excel but in order to derive meaningful results regarding their applicability this project will apply each of these two algorithms to the task of optimally playing a game of Kalah in conditions of restricted time and memory usage.

In addition to an optimal search strategy, effective gameplay also relies of a highly effective evaluation algorithm. A game of Kalah, like most real-world adversarial situations, may last for dozens of turns and so it is unrealistic to expand a tree out to the conclusion of a game in order to determine a path which reaches an ideal board state. And so it is necessary to be able to determine the optimality of a board's state at some mid-point of the game. This is the purpose of an evaluation algorithm. The code must be able to look at a board and determine its "closeness" to victory for each player.

This project has developed two evaluation algorithms whose efficacy will be compared. Each is based on its own assumptions about the flow of Kalah gameplay and has been tuned to represent these assumptions. Although these two particular algorithms are only useful in the context of Kalah, evaluation algorithms in general have very broad applicability and so it is reasonable to say that advances in the development of these algorithms is also broadly applicable. This project seeks to not only evaluate these two specific evaluation algorithms but to contribute to the process of algorithmic design and testing.

## 3.1 Domain and Goal:

The domain of this project falls into both academic and real-world situations. Academically it is interesting to compare algorithms in order to assess their speed and memory requirements. Broad assumptions can be made from comparisons of the algorithms themselves but without actually writing out an implementation it is really impossible to say which is the most effective algorithms. So much about speed and memory requirements is both hardware and implementation specific that it is important to take these steps at regular intervals in order to determine which algorithms are maintaining relevance and which can be safely mothballed.

The real-world domain includes any problem which must be approached with the assumption that an actor is not wholly determining their own conditions. An example might be shipping and transportation. In this scenario a planner can choose certain elements, like which warehouse a product ships from, but must assume that other conditions, like traffic accidents or shortages, are being determined by some adversary who will present the worst possible conditions.

The goal of this project is to evaluate the performance of four algorithms: two for search and two for evaluation. These algorithms will be compared in terms of memory usage, speed of execution, and and optimality.

## 3.2 Proposed Solution:

A total of twelve scenarios will be run and the final results will be aggregated and compared in to compare the speed and efficacy of each algorithm. The scenarios will be:

1. Vishal vs. Gentry with MinMaxAB at depth 3
2. Vishal vs. Gentry with AlphaBeta at depth 3
3. MinMax vs. AlphaBeta with Vishal at depth 3
4. MinMax vs. AlphaBeta with Gentry at depth 3
5. Vishal vs. Gentry with MinMaxAB at depth 5
6. Vishal vs. Gentry with AlphaBeta at depth 5
7. MinMax vs. AlphaBeta with Vishal at depth 5
8. MinMax vs. AlphaBeta with Gentry at depth 5
9. Vishal vs. Gentry with MinMaxAB at depth 7
10. Vishal vs. Gentry with AlphaBeta at depth 7
11. MinMax vs. AlphaBeta with Vishal at depth 7
12. MinMax vs. AlphaBeta with Gentry at depth 7

Playing these twelve games will give us six scenarios in which the Vishal algorithm and Gentry algorithm and six scenarios in which MinMaxAB is pitted against AlphaBeta Search. By aggregating the wins, node visited, and time of execution of all 12 games we can determine which search and which evaluation algorithm performs the best, which is fastest, and which visits the fewest nodes.

# 4. Evaluation Function Design:

The code for both evaluation algorithms is presented later in this document. The algorithmic design of the Vishal is presented in his own documentation. This section will focus of the Gentry evaluation algorithm and its design.

The Gentry evaluation algorithm is built on a short list of observations of conditions which should be sought by a Kalah player. These conditions are:

1. Keeping the right hand cup empty is advantageous.
2. Empty holes are as good as their neighbor is full.
3. Series of "move again" holes are advantageous.
4. Having fewer stones on  player's side is good for the player.
5. Having more stones in a player's Kalah is good.

These observations are not derived from the rules of the game but rather were gleaned from many rounds of playing Kalah and observing which conditions contributed most to a when. Broadly these conditions allow a player to maintain control of the board and help set up high value captures by cultivation empty holes or by emptying all of the stones on the player's side of the board.

A weight is given to each condition in the algorithm's implementation. This weight allowed the coders to tune the algorithms performance based on its previous performances. Scoring was given the highest weight for trivial reasons, followed by empty ups and "move again" cups being equally weighted, and finally the fewer stones and empty right hand cup were given the lowest significance in the algorithm. By checking for these conditions and applying these weights a value was assigned to each leaf node in the game tree.

# 5. Search Algorithm:

The purpose of any search algorithm is to find an optimal path through some graph of options. An internet search algorithm maps search strings onto webpages. A pathfinding searches founds routes from one location to another. Adversarial searches seek to find optimal paths through trees in which a player is not the only one making decisions on the conditions of a game (where a game can be an adversarial situation). Min-max searching seeks to find the “best of the worst” meaning that it delivers an optimal solution from amongst the worst conditions chosen by a player’s adversary.

This project has evaluated the performance of two common search algorithms: MinMaxAB and Alpha Beta Search. Each of these two seeks to perform optimised min-max searching in conditions of limited time and memory by selectively eliminating branches which cannot return a result better than what has already been found. This is effected by storing a best solution and recognizing conditions under which no better condition can be returned by a branch based on the assumption that an adversary will always choose the “worst” conditions for a player.

The two algorithms chosen for searching by this project are MinMaxAB and AlphaBeta Search. Their specifics are discussed below.

## 5.1 MinMax AB:

## 5.2 Alpha Beta Search:

# 6. Methodology:

The Kalah game was built using object oriented programming in accordance with best practices as determined by the industry. This allows each element of the game to store the data which it needs independently and to control the other objects' access to that data. A driver function was written to conduct the execution of the objects within the program.

Since the intent of this project was to compare the performance of varying search and evaluation algorithms, two seperate game trees were maintained throughout the execution of the program. Each tree was assigned its own search and its own evaluation algorithm. By re-assigning these algorithms in successive iterations of the program we were able to collect data about the performance of each algorithm.

The game objects were built so that the rules of the game were embedded in the lowest-level objects and the higher level-objects needn't track the legality of a move at their level. Rather they relied on a function provided by the lower level objects to determine the legality of a move. In this way the higher level objects and the drive function could be more game independent. This means that the focus could be placed entirely on effective searching and that the code will be easily reusable in later projects.

The driver function only needed access to the higher level objects and was therefore written entirely independently of the functionality of Kalah. This code could easily be re-implemented with any simple two player game.

All of the data on the algorithms performance is collected in the driver function. That data is reported so that it can be collected and analyzed.

## 6.1 Data Structures:

The code base of this project has mostly been separated into two objects: Board and Tree. The specific implementation of these objects is presented later in this document. Each of the two objects were structured so as to only have access to those data objects which were necessary at their level of execution.

The Board object holds one game board. Its data members are:

- The number of stones in each hole and kalah
- The boards value
- Whether the board represents a won game
- Static constants for the number of holes and stones.

Additionally the Board object is able to determine which moves are legal to play on that board based on the hard-coded rules of the games. The Board will raise an error if any other object attempts to make a move which is not legal to play. A Board's value can be determined in two ways. The first is used for leaf nodes in the game tree and that is two use one of the two evaluation algorithms provided by this project. The second is to pass the Board a value. This is used for parent nodes in the tree.

The Tree object represents a collection of Boards. The root node of the Tree is the current state of the game and all of its children represent a possible, legal state of the board. The Tree object assures that only legal states are included in the tree. The data members of the Tree are:

- A flag indicating which search algorithm to apply.
- A flag indicating which evaluation algorithm to apply to the leaf nodes of the game tree.
- The maximum depth that the tree should be built to.
- The total number of Boards in the tree.
- A flag indication which player is represented by the tree.
- An array of Boards representing the game tree.

Searching is solely the responsibility of the Tree object. It builds its tree, assigns a value to each node, and then chooses an ideal move each time the play method is called by the driver function. Tree is ignorant of the rules of the game and instead relies on the Board object to determine if a move is legal.

# 7. Program Implementation:

This program was written in C++ using the Code::Blocks IDE. All sample data was collected on code compiled using the GCC compiler suite and run on a home PC running Ubuntu 18 with an Intel I7. GPU execution was not used during the test runs of this code. The results should be taken as relative rather than absolute as the code will perform differently on different platforms.

## 7.1 Header Files:

```
#ifndef BOARD_H
#define BOARD_H


class Board
{
   public:
      //construstors and destructors
      Board();
      Board(const Board&);
      ~Board();

      //getters and setters
      int getNumHoles();
      int getAScore();
      int getBScore();
      int getValue();
      void setValue(int);
      void setValue(char, int);
      bool getFinished();

      //operator overloading
      Board& operator=(const Board&);
      bool operator==(const Board&);

      //is a move legal for a certain player?
      bool isLegal(char, int);

      //allter the board in a legal fashion
      char makeMove(char, int);

      //draw the current board on the console
      void drawBoard();

      //returns the character of the opposite player
      static char switchPlayer(char);
```

```
    private:
        //The number of holes for each player
        static const int NUM_HOLES = 6;
        //The number of stones in each hole at start
        static const int NUM_STONES = 6;

        //let A[0] be across from B[NUM_HOLES]
        //let A[0] be closest to A's kalah
        //let B[0] be closest to B's kalah
        int playerAHoles[NUM_HOLES];
        int playerBHoles[NUM_HOLES];

        //A player's score is also the number of stones in his kalah
        int playerAScore;
        int playerBScore;

        //The value of the board is calculated with the setValue func
        int value;

        //true when the board is finished
        bool finished;


        //called by setValue
        void gentryValue(char);
        void vishalValue(char);
};

#endif // BOARD_H
```

```
#ifndef TREE_H
#define TREE_H

#include "Board.h"


class Tree
{
   public:
      //constructors and destructors
      Tree();
      Tree(int, int, int, char);
      virtual ~Tree();

      //getters and setters
      bool getFinished();
      int getAScore();
      int getBScore();
      int getTotalBoards();

      //play function updates tree
      void play(char&, char&, int&);
      //draw the root board in tree
      void draw();

      static const int MINMAXAB = 1;
      static const int ALPHABETA = 2;
      static const int GENTRY = 1;
      static const int VISHAL = 2;
      static const int FIRST_MOVE = 'F';


   private:
      int searchAlg;
      int valueAlg;
      int maxDepth;
      int totalBoards;
      char player;
      Board * boards;
```

```
        static const int NUM_HOLES = 6;

        int getChild(int, int);
        int getParent(int);
        int getTier(int);

        void buildTree();
        int alphabeta(int, int, char, int, int);
        int minMaxAB(int, int , char , int , int);

        int chooseBestMove();
};

#endif // TREE_H
```

## 7.2 Source Files:

```
#include "Board.h"
#include <iostream>

using namespace std;

//Default Constructor
//Every hole should have the starting number of stones
Board::Board(){
   for(int i = 0; i < NUM_HOLES; i++){
      playerAHoles[i] = NUM_STONES;
      playerBHoles[i] = NUM_STONES;
   }

   playerAScore = 0;
   playerBScore = 0;

   value = -9999;
   finished = false;
}

//Copy Constructor
//The new board should have all the values of the old
Board::Board(const Board& toCopy){
   for(int i = 0; i < NUM_HOLES; i++){
      playerAHoles[i] = toCopy.playerAHoles[i];
      playerBHoles[i] = toCopy.playerBHoles[i];
   }

   playerAScore = toCopy.playerAScore;
   playerBScore = toCopy.playerBScore;

   value = toCopy.value;
   finished = toCopy.finished;
}
```

```
Board::~Board(){return;}

char Board::switchPlayer(char oldPlayer){
   if (oldPlayer == 'A') return 'B';
   else if (oldPlayer == 'B') return 'A';
   else
      cerr << "Bad player value passed to switchPlayer" << endl;
      return 'C';
}

//Use Gentry's algorithm to set the value of the board for a player
void Board::gentryValue(char player){
   //Player A = bottom player
   //Player B = top player

   //Tuning Parameters
   int emptyRightScaling = 1;
   int emptyCupScaling = 2;
   int scoreScaling = 5;
   int moveAgainCupsScaling = 2;
   int moreStonesScaling = 1;

   int sumA = 0, sumB = 0;
   value = 0;

   //calculate a score for Player A

   //keeping right hand hole empty is good
   if (playerAHoles[0] == 0) value += emptyRightScaling;
   if (playerBHoles[0] == 0) value -= emptyRightScaling;

   //an empty cup should add the value of the oposite one
   for (int i = 0; i < NUM_HOLES; i++){
      if (playerAHoles[i] == 0) value += playerBHoles[NUM_HOLES - i] * emptyCupScaling;
      if (playerBHoles[i] == 0) value -= playerBHoles[NUM_HOLES - i] * emptyCupScaling;
   }

   //scoring is good
   value += playerAScore * scoreScaling;
```

```
    value -= playerBScore * scoreScaling;

    //check for "move again" cups
    for (int i = 0; i < NUM_HOLES; i++){
        if (playerAHoles[i] == (i +1))
            value += moveAgainCupsScaling;
        if (playerBHoles[i] == (i + 1))
            value -= moveAgainCupsScaling;
    }

    //having less stones on my side is good-ish
    for (int i = 0; i < NUM_HOLES; i++){
        sumA += playerAHoles[i];
        sumB += playerBHoles[i];
    }
    if (sumA < sumB) value += moreStonesScaling;
    else if (sumB < sumA) value -= moreStonesScaling;

    //negate the score if the player is B
    if (player == 'A') return;
    else if (player == 'B'){
        value = -1 * value;
        return;
    }
    else {
        cerr << "Bad player value passed to gentryValue()" << endl;
        return;
    }
}

//Use Vishal's algorithm to set the value of the board for a player
void Board::vishalValue(char player){
    int store1[5] = {0};
    int counter=0;
    int score=0;
    int GrabFromEmpty = 200;
    int MakeAscending = 100;
    int search =0;
    int playerNum;
```

```
    //So if i find an empty pit in my end and see how many stones are present in the opposite end,
    // grab all the stones in the next move;

    for(int i=0;i<NUM_HOLES;i++)
    {
       if(playerAHoles[i]==0)
       {
          store1[counter]=i;
          counter++;
       }
    }
    //cout<<"---------"<<endl;
    for(int i=0;i<counter;i++)
    {

       //cout<<store1[i]<<" "; // found empty pits in my end;
    }
    //cout<<endl<<"---------"<<endl;

    int oppvalue=0;
    int high = 0;
    for(int i=NUM_HOLES-1; i>0 ; i--)
    {
       if(playerAHoles[i]>playerAHoles[high])
          high=i;
    }
//    //cout<<endl<<"Highest value in the opp end"<<endl<<high<<endl;
//    int size = sizeof(store1)/store1[0];
//
    if (player == 'A') playerNum = 1;
    else playerNum = 2;

    switch (playerNum) {
       case 1:
          for(int i=0;i<NUM_HOLES;i++)
          {
             score=0;
             search=store1[i];
```

```
            if(search ==0)
            {
               if(playerAHoles[search+1]==12 || playerAHoles[search+2]==11 ||
playerAHoles[search+3]==10 || playerAHoles[search+4]==9||playerAHoles[search+5]==8)
               {
                  score+=GrabFromEmpty;
               }
               value = score;
               return;
            }
            else if(search ==1)
            {
               if(playerAHoles[search+1]==12 || playerAHoles[search+2]==11 ||
playerAHoles[search+3]==10 || playerAHoles[search+4]==9 || playerAHoles[search-1]==1)
               {
                  score+=GrabFromEmpty;
               }
               value = score;
               return;
            }
            else if(search ==2)
            {
               if(playerAHoles[search+1]==12 || playerAHoles[search+2]==11 ||
playerAHoles[search+3]==10||playerAHoles[search-1]==1||playerAHoles[search-2]==2)
               {
                  score+=GrabFromEmpty;
               }
               value = score;
               return;
            }
            else if(search ==3)
            {
               if(playerAHoles[search+1]==12 ||
playerAHoles[search+2]==11||playerAHoles[search-1]==1||playerAHoles[search-2]==2||playerA
Holes[search-3]==3)
               {
                  score+=GrabFromEmpty;
               }
               value = score;
```

```
                return;
            }
            else if(search ==4)
            {
                if(playerAHoles[search+1]==12 || playerAHoles[search-1]==1||
playerAHoles[search-2]==2|| playerAHoles[search-3]==3|| playerAHoles[search-4]==4)
                {
                    score+=GrabFromEmpty;
                }
                value = score;
                return;
            }
            else if(search ==5)
            {
                if(playerAHoles[search-1]==1|| playerAHoles[search-2]==2||
playerAHoles[search-3]==3|| playerAHoles[search-4]==4 || playerAHoles[search-5]==5)
                {
                    score+=GrabFromEmpty;
                }
                value = score;
                return;
            }
            //Will have to cover the other player's holes as well!

        }
        break;

      default:
        break;
   }
//
//
//   //check for the opp end ascending order and try to attack!
   value = score;
   return;
}

int Board::getNumHoles() {return NUM_HOLES;}
```

```
int Board::getAScore() {return playerAScore;}

int Board::getBScore() {return playerBScore;}

int Board::getValue() {return value;}

void Board::setValue(int newValue){
   value = newValue;
   return;
}

void Board::setValue(char player, int valueAlgorithm){
   switch(valueAlgorithm){
      case 1:
         gentryValue(player);
         break;
      case 2:
         vishalValue(player);
         break;
      default:
         cerr << "Bad value alg choice sent to setValue" << endl;
         value = 0;
         break;

      break;
   }
}

bool Board::getFinished() {return finished;}

Board& Board::operator=(const Board& toCopy){
   if (this == &toCopy) return *this;

   for (int i = 0; i < NUM_HOLES; i++){
      playerAHoles[i] = toCopy.playerAHoles[i];
      playerBHoles[i] = toCopy.playerBHoles[i];
   }

   playerAScore = toCopy.playerAScore;
```

```
    playerBScore = toCopy.playerBScore;

    value = toCopy.value;

    finished = toCopy.finished;

    return *this;
}

bool Board::operator==(const Board& toCompare){
    if (this == &toCompare) return true;

    for(int i = 0; i < NUM_HOLES; i++){
        if (playerAHoles[i] != toCompare.playerAHoles[i]) return false;
        if (playerBHoles[i] != toCompare.playerBHoles[i]) return false;
    }

    if (playerAScore != toCompare.playerAScore) return false;
    if (playerBScore != toCompare.playerBScore) return false;

    if (value != toCompare.value) return false;

    if (finished != toCompare.finished) return false;

    return true;
}

bool Board::isLegal(char player, int hole){
    //cannot move if the game is over
    if (finished) {
        //cerr << "isLegal called for finished board" << endl;
        return false;
    }

    if (player != 'A' && player != 'B'){
        //cerr << "Bad player value passed to isLegal" << endl;
        return false;
    }
```

```
    //hole must be in range 0-5
    if (hole < 0 || hole >= NUM_HOLES){
        //cerr << hole << " hole value outside of range in isLegal" << endl;
        return false;
    }

    //moving an empty hole is not legal
    if (player == 'A' && playerAHoles[hole] == 0) {
        //cerr << hole << " hole is empty in isLegal" << endl;
        return false;
    }
    if (player == 'B' && playerBHoles[hole] == 0) {
        //cerr << hole << " hole is empty in isLegal" << endl;
        return false;
    }
    return true;
}

char Board::makeMove(char player, int hole){
    int numberStones;
    int sumA = 0, sumB = 0;
    int winningScore = (NUM_HOLES * NUM_STONES + 1);
    char oldPlayer = player;

    if (!isLegal(player, hole)){
        //cerr << "Illegal move attempted in makeMove" << endl;
        //value = -999;
        return player;
    }

    //remove stones from played hole
    if (player == 'A') {
        numberStones = playerAHoles[hole];
        playerAHoles[hole] = 0;
    }
    else if (player == 'B') {
        numberStones = playerBHoles[hole];
        playerBHoles[hole] = 0;
    }
```

```
else{
   cerr << "Bad player value passed to makeMove" << endl;
   return 'C';
}
hole--;

//playing an empty cup is illegal
if (numberStones == 0){
   cerr << "An empty cup was played at makeMove" << endl;
   return 'C';
}

//keep placing stones until we reach the last stone
while (numberStones > 1){
   //Case 1: regular hole for player A
   if (player == 'A' && hole >= 0){
      playerAHoles[hole]++;
      hole--;
   }
   //Case 2: kalah for player A
   else if (player == 'A' && hole == -1){
      if (oldPlayer == 'A'){
         playerAScore++;
      }
      else {
         //do not place in an opponent's kalah
         numberStones++;
      }
      hole = NUM_HOLES-1;
      player = 'B';
   }
   //Case 3: regular hole for player B
   else if (player == 'B' && hole >= 0){
      playerBHoles[hole]++;
      hole--;
   }
   //Case 4: kalah for player B
   else if (player == 'B' && hole == -1){
      if (oldPlayer == 'B'){
```

```
            playerBScore++;
        }
        else {
            //do not place in an opponent's kalah
            numberStones++;
        }
        hole = NUM_HOLES -1;
        player = 'A';
    }
    //Case 5: something has gone wrong
    else{
        cerr << "Bad state reached placing stones in makeMove" << endl;
        return 'C';
    }

    numberStones--;
}

//Place the last stone
if (numberStones != 1){
    cerr << "Error placing last stone in makeMove" << endl;
    return 'C';
}

//Do not let the last move be the opponent's kalah
if (hole == -1 && player != oldPlayer){
    hole = NUM_HOLES - 1;
    player = switchPlayer(player);
}

//Case 1: Player A's kalah
if (player == 'A' && hole == -1){
    playerAScore++;
    //Do not switch player
}
//Case 2: Player B's kalah
else if (player == 'B'  && hole == -1){
    playerBScore++;
    //Do not switch player
```

```
}
//Case 3:Empty hole for A
else if (player == 'A' && oldPlayer == 'A' && playerAHoles[hole] == 0){
   //Add the stone of opposite cup to A's kalah
   playerAScore += playerBHoles[NUM_HOLES - hole - 1];
   playerBHoles[NUM_HOLES - hole - 1] = 0;
   //Add the final stone to A's kalh
   playerAScore++;
   //Switch player
   oldPlayer = 'B';
}
//Case 4: Empty hole for B
else if (player == 'B' && oldPlayer == 'B' && playerBHoles[hole] == 0){
   //Add the stone of opposite cup to B's kalah
   playerBScore += playerAHoles[NUM_HOLES - hole - 1];
   playerAHoles[NUM_HOLES - hole - 1] = 0;
   //Add the final stone to A's kalh
   playerBScore++;
   //Switch player
   oldPlayer = 'A';
}
//Case 5: Non-empty hole for A
else if (player == 'A' && hole >= 0){
   //Add stone to hole
   playerAHoles[hole]++;
   //switch player
   oldPlayer = switchPlayer(oldPlayer);
}
//Case 6: Non-empty hole for B
else if (player == 'B' && hole >= 0){
   //Add stone to hole
   playerBHoles[hole]++;
   //switch player
   oldPlayer = switchPlayer(oldPlayer);
}
//Case 7: error case
else {
   cerr << "Bad state reached placing last stone in makeMove" << endl;
   return 'C';
```

```
    }

    //Check to see if either side is empty
    for (int i = 0; i < NUM_HOLES; i++){
        sumA += playerAHoles[i];
        sumB += playerBHoles[i];
    }
    if (sumA == 0){
        playerAScore += sumB;
    }
    if (sumB == 0){
        playerBScore += sumA;
    }

    //Check for win condition
    if (playerAScore >= winningScore || playerBScore >= winningScore){
        finished = true;
    }

    //Complete
    return oldPlayer;

}

void Board::drawBoard(){
    cout << "\t";
    for (int i = 0; i < NUM_HOLES; i++)
        cout << playerBHoles[i] << " ";
    cout << endl;
    cout << playerBScore << "\t\t\t\t" << playerAScore << endl;
    cout << "\t";
    for (int i = NUM_HOLES -1; i >= 0; i--)
        cout << playerAHoles[i] << " ";
    cout << endl;

    return;
}
```

```
#include "Tree.h"
#include <math.h>
#include <iostream>

using namespace std;

//Default constructor. Should not be used
Tree::Tree(){
   totalBoards = 7;
   boards = new Board[totalBoards];
   player = 'A';
   maxDepth = 1;
   searchAlg = MINMAXAB;
   valueAlg = GENTRY;

   buildTree();
}

Tree::Tree(int searchAlg, int valueAlg, int maxDepth, char player){
   this->searchAlg = searchAlg;
   this->valueAlg = valueAlg;
   this->maxDepth = maxDepth;
   this->player = player;

   totalBoards = 0;
   for (int i = 0; i <= maxDepth; i++)
      totalBoards += pow(NUM_HOLES, i);
   //cout << "Total boards in new tree: " << totalBoards << endl;

   boards = new Board[totalBoards];
   buildTree();
}
Tree::~Tree(){
   delete[] boards;
}

int Tree::getTier(int boardNumber){
   int tier = 0;
   int total = 1;
```

```
    while (boardNumber >= total){
        tier++;
        total += pow(NUM_HOLES, tier);
    }
    //cout << "Child " << boardNumber << " is on tier " << tier << endl;
    return tier;
}

int Tree::getChild(int current, int moveNum){
    int child = (6*current + moveNum + 1);
    if (child >= totalBoards){
        cerr << "Child produced in getChild is out of bounds" << endl;
    }
    return child;
}

int Tree::getParent(int current){
    current--;
    return (current / 6);
}

//node 0 is the root and should be in a position determind in play()
//every child node is a legal move from its parent
//an illegal child duplicates its parent
void Tree::buildTree(){
    int currentNode = 1;
    int currentDepth;
    while (currentNode < totalBoards){
        currentDepth = getTier(currentNode);
        for (int i = 0; i < NUM_HOLES; i++){
            boards[currentNode] = boards[getParent(currentNode)];
            if (boards[currentNode].isLegal(player, i)){
                boards[currentNode].makeMove(player, i);
            }
            if (currentDepth == maxDepth){
                boards[currentNode].setValue(player, valueAlg);
                //cout << "Leaf node: " << boards[currentNode].getValue() << endl;
            }
            else
```

```
            boards[currentNode].setValue(-9999);
        currentNode++;
        if (currentNode == totalBoards+1){
            cerr << "Out of bounds board reached in buildTree" << endl;
            break;
        }
      }
   }

   switch (searchAlg){
      case MINMAXAB:
         minMaxAB(0, 0, player, 99999, -99999);
         break;
      case ALPHABETA:
         alphabeta(0, 0, player, 99999, -99999);
         break;
      default:
         cerr << "Bad search algorithm number in buildTree()" << endl;
         break;
   }
   return;
}

//getters and setters
bool Tree::getFinished() {return boards[0].getFinished();}
int Tree::getAScore() {return boards[0].getAScore();}
int Tree::getBScore() {return boards[0].getBScore();}
int Tree::getTotalBoards() {return totalBoards;}

//draw the root board in tree
void Tree::draw() {
   boards[0].drawBoard();
   return;}

int Tree::minMaxAB(int currentNode, int depth, char player, int useThresh, int passThresh) {
   int newValue;
   char resultSucc;
   int structure;
```

```
    //check for bad values
    if (currentNode >= totalBoards){
        cerr << "Bad current node for minMaxAB" << endl;
        return -9999;
    }
    if (depth > maxDepth){
        cerr << "Bad depth value in minMaxAB" << endl;
        return -9999;
    }

    //if deep enough
    if(depth == maxDepth)
    {
        //cout << "Leaf node in MMAB. Value: " << boards[currentNode].getValue() << endl;
        return boards[currentNode].getValue();
    }

    //otherwise
    for(int i = 0 ; i < NUM_HOLES; i++)
    {
        structure = minMaxAB(getChild(currentNode, i),depth+1,
Board::switchPlayer(player),-passThresh,-useThresh);
        newValue = -structure;

        if(newValue > passThresh)
        {
            //index->set_value(i);
            boards[currentNode].setValue(newValue);
            passThresh = newValue;
        }
        if(passThresh >= useThresh)
        {
            structure=passThresh;
            return structure;
        }
    }
    structure=passThresh;
    return structure;
}
```

```
int Tree::alphabeta(int currentNode, int depth, char player, int alpha, int beta)
{
   //check for bad values
   if (currentNode >= totalBoards){
      cerr << "Bad current node for alphabeta" << endl;
      return -9999;
   }
   if (depth > maxDepth){
      cerr << "Bad depth value in alphabeta" << endl;
      return -9999;
   }

   //node is a leaf node)
   if (depth == maxDepth)
      return boards[currentNode].getValue();

   if (player == this->player )
   {
      int bestVal = -100,value;
      for(int i=0;i < NUM_HOLES ; i++ )
      {
         value = alphabeta(getChild(currentNode, i), depth+1, player, alpha, beta);
         bestVal = ( bestVal > value) ? bestVal : value;
         alpha = ( alpha > bestVal) ? alpha : bestVal;
         if (beta <= alpha){
            //cout << "Beta is less than alpha" << endl;
            break;
         }
      }
      //node->set_value(bestVal);
      boards[currentNode].setValue(bestVal);
      return bestVal;
   }
   else
   {
      int bestVal = +100,value;
      for(int i=0; i < 6 ; i++ )
      {
```

```
            value = alphabeta(getChild(currentNode, i), depth+1, player, alpha, beta);
            bestVal = ( bestVal < value) ? bestVal : value;
            beta = ( beta < bestVal) ? beta : bestVal;
            if (beta < alpha)
               break;
         }
         //node->set_value(bestVal);
         boards[currentNode].setValue(bestVal);
         return bestVal;
      }
}

int Tree::chooseBestMove(){
   int bestMove;
   int bestVal = -999999;
   for (int i = NUM_HOLES; i > 0; i--){
      if (!boards[0].isLegal(player, i-1))
         continue;
      if (boards[i].getValue() > bestVal){
         bestMove = i;
         bestVal = boards[i].getValue();
      }
      //cout << "Best move: " << bestVal << endl;
   }

   bestMove -= 1;
   //cout << "My best move is " << bestMove << endl;

   //final check that bestMove is a legal move
   //if (!boards[0].isLegal(player, bestMove))
   //    return 0;
   return bestMove;
}

void Tree::play(char& currentPlayer, char& lastPlayer, int& lastMove){
   //cout << "Current player is " << currentPlayer << endl;
   //cout << "Last player is " << lastPlayer << endl;

   //Check for bad input
```

```
if (lastMove < 0 || lastMove >= NUM_HOLES){
   cerr << "Bad value for lastMove in play" << endl;
}
if (currentPlayer != 'A' && currentPlayer != 'B'){
   cerr << "Bad currentPlayer value in play" << endl;
}

//Case 1: first move for A
if (lastPlayer == FIRST_MOVE){
   //cout << "First move" << endl;
   lastPlayer = currentPlayer;
   lastMove = chooseBestMove();
   currentPlayer = boards[0].makeMove(player, lastMove);
}
//Case 2: this player's move not after a repeat
//update board then choose move then update board
else if (currentPlayer == player && lastPlayer != currentPlayer){
   //cout << player << "'s move and " << lastPlayer << " went last time." << endl;
   boards[0].makeMove(lastPlayer, lastMove);
   buildTree();
   lastPlayer = currentPlayer;
   lastMove = chooseBestMove();
   currentPlayer = boards[0].makeMove(player, lastMove);
   buildTree();
}
//Case 3: this players move on repeat
//choose move then update board
else if (currentPlayer == player && lastPlayer == currentPlayer){
   //cout << player << "'s move and I'm going again" << endl;
   lastPlayer = currentPlayer;
   lastMove = chooseBestMove();
   currentPlayer = boards[0].makeMove(player, lastMove);
   buildTree();
}
//Case 4: not this player's move
//update board without choosing move
else if (currentPlayer != player){
   //cout << "I just need to update" << endl;
   boards[0].makeMove(lastPlayer, lastMove);
```

```
        buildTree();
    }
    //Case 5: error condition
    else{
        cerr << "Bad case in play" << endl;
    }
    return;
}
```

## 7.3 Driver Function:

```
#include <iostream>
#include <cstdio>

#include "Tree.h"

using namespace std;

int main()
{
   Tree * tree1, * tree2;
   char lastPlayer, currentPlayer;
   int lastMove, aScore, bScore, depth, turn;

   //Case 1: Vishal vs. Gentry with MinMaxAB at depth 3
   cout << "***************************" << endl;
   cout << "Vishal vs. Gentry with MinMaxAB at depth 3" << endl;
   depth = 3;
   tree1 = new Tree(Tree::MINMAXAB, Tree::GENTRY, depth, 'A');
   tree2 = new Tree(Tree::MINMAXAB, Tree::VISHAL, depth, 'B');
   currentPlayer = 'A';
   lastPlayer = Tree::FIRST_MOVE;
   lastMove = 0;
   turn = 0;
   aScore = 0;
   bScore = 0;

   while(! tree1->getFinished() && ! tree2->getFinished()){
      tree1->play(currentPlayer, lastPlayer, lastMove);
      tree2->play(currentPlayer, lastPlayer, lastMove);
      turn++;
      //tree1->draw();
      //tree2->draw();
      //getchar();
   }

   aScore = tree2->getAScore();
```

```
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "Gentry wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "Vishal wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();
//Case 2: Vishal vs. Gentry with AlphaBeta at depth 3

cout << "**************************" << endl;
cout << "Vishal vs. Gentry with AlphaBeta at depth 3" << endl;
depth = 3;
tree1 = new Tree(Tree::ALPHABETA, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}
```

```
aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "Gentry wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "Vishal wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 3: MinMax vs. AlphaBeta with Vishal at depth 3
cout << "**************************" << endl;
cout << "MinMax vs. AlphaBeta with Vishal at depth 3" << endl;
depth = 3;
tree1 = new Tree(Tree::MINMAXAB, Tree::VISHAL, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
```

```
    //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
    cout << "MinMaxAB wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
    cout << "AlphaBeta wins " << bScore << " to " << aScore << endl;
}
else {
    cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 4: MinMax vs. AlphaBeta with Gentry at depth 3
cout << "**************************" << endl;
cout << "MinMax vs. AlphaBeta with Gentry at depth 3" << endl;
depth = 3;
tree1 = new Tree(Tree::MINMAXAB, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::GENTRY, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
    tree1->play(currentPlayer, lastPlayer, lastMove);
    tree2->play(currentPlayer, lastPlayer, lastMove);
    turn++;
```

```
    //tree1->draw();
    //tree2->draw();
    //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
    cout << "MinMaxAB wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
    cout << "AlphaBeta wins " << bScore << " to " << aScore << endl;
}
else {
    cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 5: Vishal vs. Gentry with MinMaxAB at depth 5
cout << "**************************" << endl;
cout << "Vishal vs. Gentry with MinMaxAB at depth 5" << endl;
depth = 5;
tree1 = new Tree(Tree::MINMAXAB, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::MINMAXAB, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
    tree1->play(currentPlayer, lastPlayer, lastMove);
```

```
    tree2->play(currentPlayer, lastPlayer, lastMove);
    turn++;
    //tree1->draw();
    //tree2->draw();
    //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
    cout << "Gentry wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
    cout << "Vishal wins " << bScore << " to " << aScore << endl;
}
else {
    cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 6: Vishal vs. Gentry with AlphaBeta at depth 5
cout << "**************************" << endl;
cout << "Vishal vs. Gentry with AlphaBeta at depth 5" << endl;
depth = 5;
tree1 = new Tree(Tree::ALPHABETA, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;
```

```
while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "Gentry wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "Vishal wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 7: MinMax vs. AlphaBeta with Vishal at depth 5
cout << "**************************" << endl;
cout << "MinMax vs. AlphaBeta with Vishal at depth 5" << endl;
depth = 5;
tree1 = new Tree(Tree::MINMAXAB, Tree::VISHAL, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
```

```
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "MinMaxAB wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "AlphaBeta wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 8: MinMax vs. AlphaBeta with Gentry at depth 5
cout << "**************************" << endl;
cout << "MinMax vs. AlphaBeta with Gentry at depth 5" << endl;
depth = 5;
tree1 = new Tree(Tree::MINMAXAB, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::GENTRY, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
```

```
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "MinMaxAB wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "AlphaBeta wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 9: Vishal vs. Gentry with MinMaxAB at depth 7
cout << "**************************" << endl;
cout << "Vishal vs. Gentry with MinMaxAB at depth 7" << endl;
depth = 7;
tree1 = new Tree(Tree::MINMAXAB, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::MINMAXAB, Tree::VISHAL, depth, 'B');
//cout << "Total tree size is: " << tree1->getTotalBoards() << endl;
```

```
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "Gentry wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "Vishal wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 10: Vishal vs. Gentry with AlphaBeta at depth 7
cout << "**************************" << endl;
cout << "Vishal vs. Gentry with AlphaBeta at depth 7" << endl;
depth = 7;
```

```
tree1 = new Tree(Tree::ALPHABETA, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "Gentry wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "Vishal wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();

//Case 11: MinMax vs. AlphaBeta with Vishal at depth 7
cout << "**************************" << endl;
```

```
cout << "MinMax vs. AlphaBeta with Vishal at depth 7" << endl;
depth = 7;
tree1 = new Tree(Tree::MINMAXAB, Tree::VISHAL, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::VISHAL, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "MinMaxAB wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "AlphaBeta wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
getchar();
//Case 12: MinMax vs. AlphaBeta with Gentry at depth 7
```

```
cout << "**************************" << endl;
cout << "MinMax vs. AlphaBeta with Gentry at depth 7" << endl;
depth = 7;
tree1 = new Tree(Tree::MINMAXAB, Tree::GENTRY, depth, 'A');
tree2 = new Tree(Tree::ALPHABETA, Tree::GENTRY, depth, 'B');
currentPlayer = 'A';
lastPlayer = Tree::FIRST_MOVE;
lastMove = 0;
turn = 0;
aScore = 0;
bScore = 0;

while(! tree1->getFinished() && ! tree2->getFinished()){
   tree1->play(currentPlayer, lastPlayer, lastMove);
   tree2->play(currentPlayer, lastPlayer, lastMove);
   turn++;
   //tree1->draw();
   //tree2->draw();
   //getchar();
}

aScore = tree2->getAScore();
bScore = tree2->getBScore();

cout << turn << " turns played" << endl;

if (aScore > bScore){
   cout << "MinMaxAB wins " << aScore << " to " << bScore << endl;
}
else if (bScore > aScore){
   cout << "AlphaBeta wins " << bScore << " to " << aScore << endl;
}
else {
   cout << "Tie game." << endl;
}

delete tree1;
delete tree2;
```

```
    return 0;
}
```