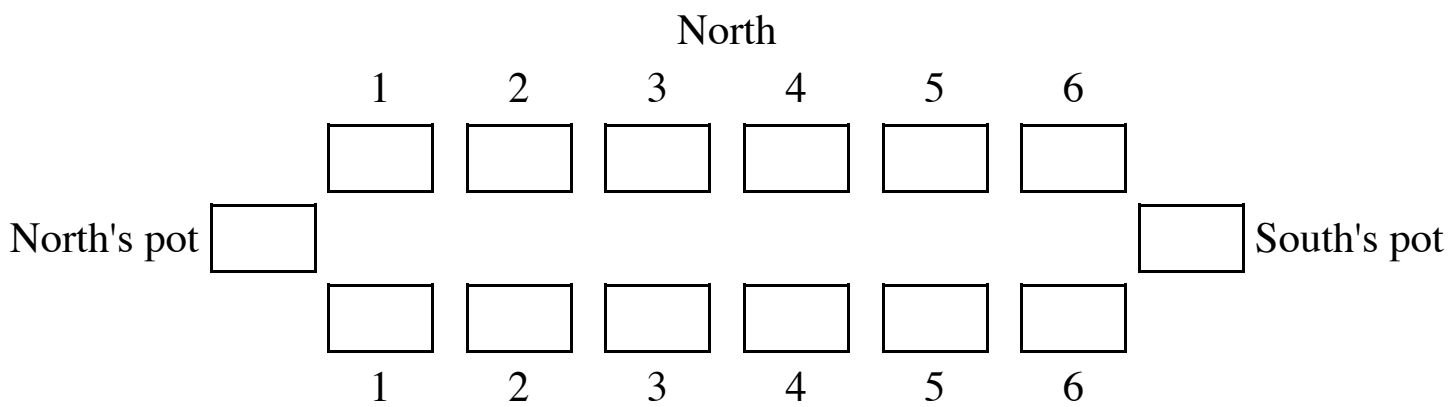
Programming Assignment 3
Kalah

Time due: 11:00 PM Tuesday, May 21

Kalah is one of a family of African and Asian games that have been played since ancient times. For this project, you are to write a program that plays the game of Kalah well. The game has many variants, so we'll first describe the one we'll use for this project.

Kalah is played on a board that looks like this:



Each of the two players, North and South, has six holes on his side of the board and one pot (his "kalah"). Each hole start off with four beans in it. Players take turns making moves. A move begins with a player picking up all of the beans in one of his six holes. Then, proceeding counterclockwise, he puts one bean in each hole and his own pot (skipping over his opponent's pot if the "sowing" gets that far), until all of the beans he picked up have been sown. For example, if South had 4 beans in his #5 hole, he might pick them up and deposit one each in his #6 hole, his pot, and North's #6 and #5 hole. Depending on where the last bean ends up, one of three things happens:

- If it was placed in the player's pot, he must take another turn.
- If it was placed in one of the player's own holes that was empty just a moment before, and if the opponent's hole directly opposite from that hole is not empty, then that bean and all beans in the opponent's hole directly opposite from that hole are put into the players pot, and the turn ends. (This is a *capture*.) Notice that it is not a capture if the player's last bean ends up in what was an empty hole on the opponent's side.
- In all other cases, the turn ends.

Whenever a turn ends with all of the holes on one side of the board empty (no matter whose turn ended), the game is over, and any beans in the other player's holes are put into that other player's pot. The winner is then the player with the most beans in his pot.

As an example of the rules, consider the following endgame; it's South's turn:

```
                North
              1  2  3  4  5  6
              ---------------
              2  4  0  0  0  0
North's pot  22                 16  South's pot
              0  0  1  1  1  1
              ---------------
              1  2  3  4  5  6
                South
```

Suppose South moves from her hole #6:

```
              2  4  0  0  0  0
           22                 17
              0  0  1  1  1  0
```

Since that ended up in her pot, she goes again, choosing, say, hole #4:

```
              2  4  0  0  0  0
           22                 17
              0  0  1  0  2  0
```

Suppose North now chooses his hole #1:

```
              0  4  0  0  0  0
```

```
23                      17
    1   0   1   0   2   0
```

If South now chooses her hole #1, she captures the beans in North's hole #2:

```
    0   0   0   0   0   0
23                      22
    0   0   1   0   2   0
```

Since all the holes on one side are empty, the game is over, and the 3 beans still in South's holes go into South's pot:

```
    0   0   0   0   0   0
23                      25
    0   0   0   0   0   0
```

South has 25 beans to North's 23, so South wins.

Your program will be designed to allow two players to play against each other. Each player will be either a human or a computer player. The computer player's behavior will be embodied in a C++ class described later. You will write different classes for different kinds of computer players. For example, one kind of computer player may be really dumb and just pick an arbitrary one of its holes for its turns. Another may play by considering moves and countermoves, selecting the move it determines is best. Let's see how it might do that.

How to play intelligently

Until you are ready to implement the SmartPlayer::chooseMove function described later, you can get by with just skimming this section and continue reading at the Your assignment *section*.

Game playing is one area in the field of computer science called artificial intelligence (making computers do tasks which appear to require human reasoning ability). What you want your program to do is somehow model what a human does when he or she plays: Consider the possible moves (and the opponent's countermoves, and the replies to those countermoves, etc.) and select one that is in some way best.

To accomplish this, a program needs to be able to do three things:

- Determine what all the legal moves are from a given board configuration, including determining whether the game is finished.
- Decide just how "good" a particular board configuration is for a player.
- Organize the previous two tasks so that possible good moves are not overlooked and time is not wasted considering many useless moves.

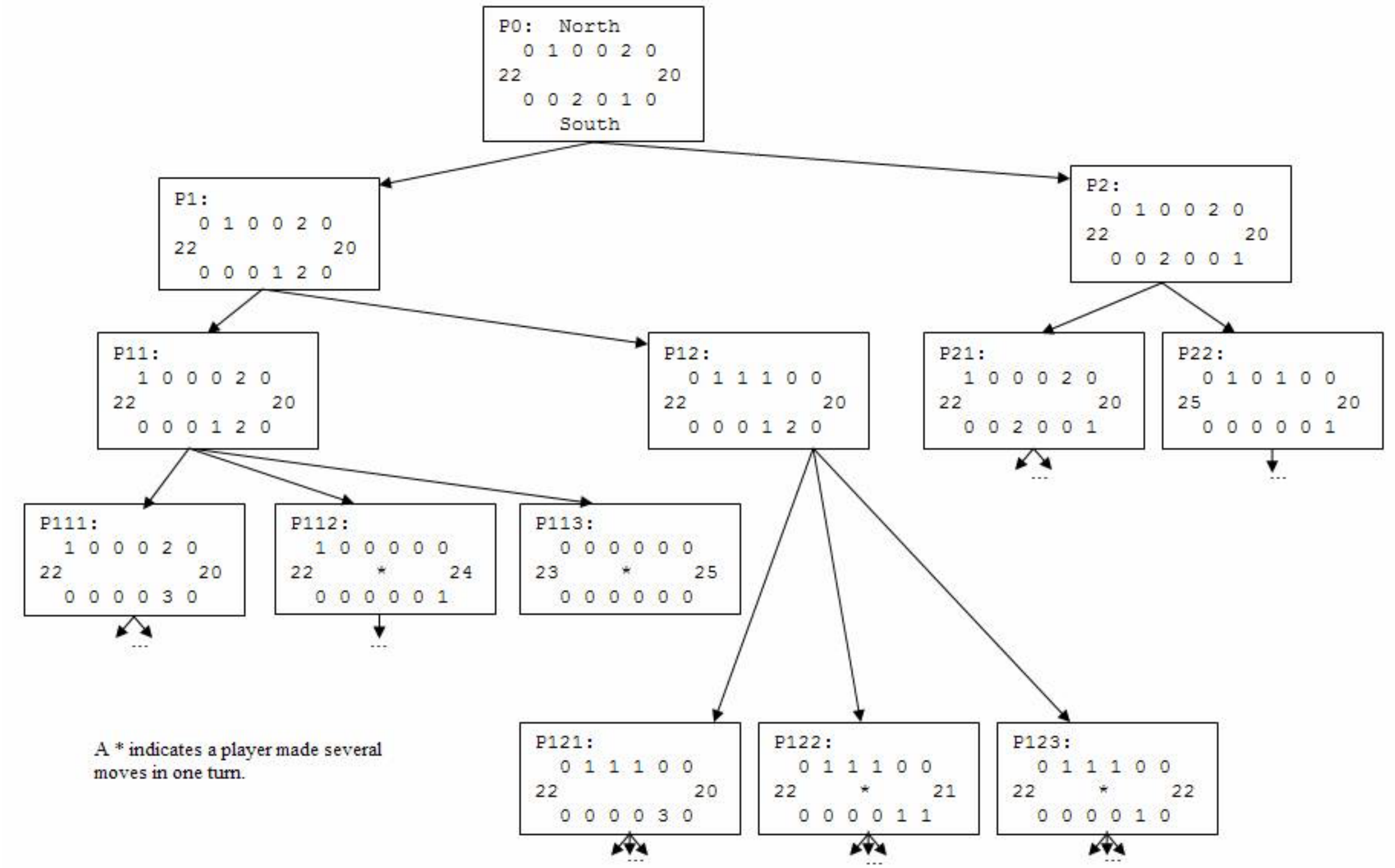
Because the rules of Kalah are simple, the first task (determining legal moves) should be easy to program. We will hold off discussing the second task until we talk about the third. For the third task, we will use a *game tree* to organize in our minds (and implicitly or explicitly in your program) the task of considering sequences of moves.

Game trees

We can represent the possible positions that can develop from a given one by drawing a tree (rather like the macOS or Windows tree of folders for files). At the top (the *root* of the tree, we show the given position. Beneath it, we draw the positions resulting from all of the possible moves that the player whose turn it is can make. Below each of those positions, we draw the positions resulting from each possible opponent countermove to the move that resulted in that position. We can continue this as long as we want, subject to time and space limitations. For example, from the position

```
          North
    0   1   0   0   2   0
22                      20
    0   0   2   0   1   0
          South
```

with South to move next, we can derive this (partial) game tree:



A * indicates a player made several moves in one turn.

Notice that the nodes one level down from the root reflect the results of South's possible moves. Two levels down are the results of North's countermoves. Three levels down are South's replies to those countermoves. (Boredom kept us from drawing further levels of the tree.)

(Notice, by the way, what South did in position P11 to result in P112: First, sow the two beans from hole #5, ending in the pot and getting another turn. Then, sow from hole #4, capturing the beans in North's hole #5. To get from P11 to P113, first sow from hole #5, then #6, then #4. Since the capture left no beans on South's side, the game is over and North's remaining bean is swept into North's pot.)

Given the *complete* game tree with a given position at the root, a player can decide the best move to make by finding all the paths through the tree that lead to a winning position and choosing a move, if any, that is on a winning path that the opponent cannot force him from without putting him on another winning path. Unfortunately, the complete game tree developed from a position is usually too large to traverse in a reasonable amount of time. Instead, we limit ourselves to "looking ahead" only a few moves (i.e., examining only a few levels of the game tree). Since there may not be a winner after only a few moves, we need other criteria to decide how good a position is.

Evaluation functions

We will return to game trees after we discuss the second task mentioned above: an *evaluation function* that, given a position, returns a number that measures how good that position is for the player.

A human evaluates a position by applying various rules of thumb to it. For example, "The more beans I have in my pot, the better" is a simple rule of thumb for Kalah, but "The greater the difference between the number of beans in my pot and the number in my opponent's, the better" is probably a better one. These rules of thumb are called *heuristics*. Our evaluation functions should reflect the heuristics we believe are good, perhaps giving more weight to some more than others.

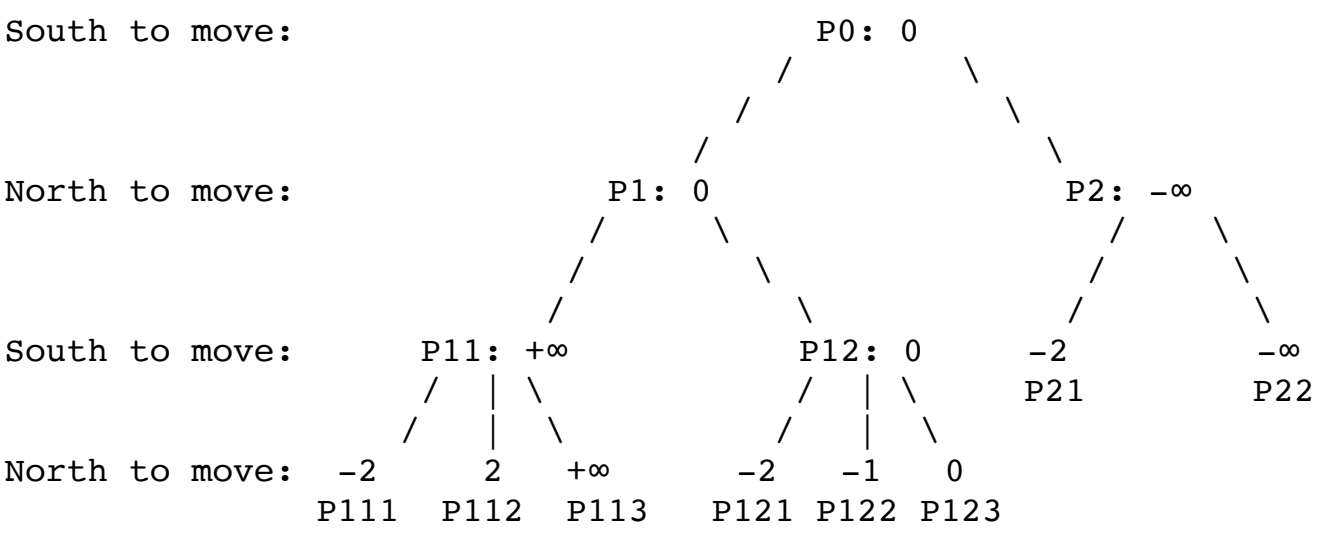
For many two-player games, Kalah among them, the same evaluation function that measures how good a position is for one player also measures how bad that position is for the other. If the function gives high numbers to positions that are good for South, then South will try to get into positions that evaluate to high numbers, while North will try to move into positions rated low by the same function (since bad for South is good for North).

The minimax algorithm

The minimax algorithm is a way for your program to search the game tree so that it always finds what it considers to be the best play

available. Minimax works on the assumption that each player will make the best move available to him at the time, where "best" is measured using the evaluation function. A player can never be hurt by an opponent playing less than optimally.

To decide on a move from a given position, we generate a game tree for that position, cutting it off at some point. We then apply the evaluation function to each leaf of the tree (i.e., each node from which we did not generate any further nodes). As an example, let's use the tree shown above, assuming that we cut off the tree generation at the nodes shown. For our evaluation function, we'll use "the number of beans in South's pot minus the number of beans in North's pot", except that a winning position for South is valued at +∞ (or a very large number), and a loss for South is −∞, since winning and losing are the best and worst things that can happen. Then the values of the positions in the tree are as follows (we'll explain how we arrived at the numbers in the non-leaf nodes):



(Notice that P22 is valued at −∞, since North's having a majority of the beans makes it impossible for South to win.)

Now, working from the leaves of the tree toward the root, we fill in the values for the non-leaf nodes according to these rules:

- If the transition from the non-leaf node under consideration to the next level down represents South's move, then assign to that node the *maximum* of the values from the lower level. (This means that South will always make the move that leads to the best outcome for South, assuming perfect play by North; if North messes up, South can do no worse, and might possibly even do better.) For example, node P12 was assigned the 0 from P123, since the -2 of P121 and the −1 of P122 would be worse for South.)

- If the transition represents North's move, assign the *minimum* of the values from below. (This means that North will always make the move that leads to the worst outcome for South.) For example, P1 is 0 because P12 won't lead to South's winning, and P11 will.

Once we have a value for the root node, we make the move that corresponds to that value (in the example, the move that leads to P1). If there is more than one move leading to that value, then any will do (although we might try to decide which one would give our opponent the greatest possibility of making an error.)

Here's high-level pseudocode for a function that determines a player's best move. Its input parameters are the player whose move it is (i.e. South or North) and the board state to be considered; its output parameters are the best hole to sow from for the move and the value that can be achieved if that move is made. If the game is over, the best hole is set to −1. Also, since the entire game tree is too big to search completely, we'll have some criterion that causes the search not to explore the nodes below certain nodes; instead, we just evaluate that board position and set the best hole to −1. The criterion could be based on how deep the search has gone, how much time has elapsed, and/or some other factors.

```
void chooseMove(in: player, board; out: bestHole, value):
        if no move for player exists (i.e., game is over),
                bestHole = -1
                value = value of this position (i.e., +∞, -∞, or 0)
                return
        if the criterion says we should not search below this node
                bestHole = -1
                value = value of this position
                return
        for every hole h the player can choose
                "make" the move h
                chooseMove(opponent, h2, v2)
                "unmake" the move
                if v2 is better for the player than best seen so far,
                        bestHole = h
                        value = v2
        return
```

Again, this is high-level pseudocode, so you'll have to decide on implementation details. For example, if the criterion is based on the depth of the search, you'll probably need another input parameter representing the search depth. To "make" and "unmake" moves you're considering without moving for real, you may want to work with copies of the board.

Your assignment

For this project, you will write several classes that will work together to play Kalah. So that we can effectively test your program, we will specify some classes you must have and public functions they must implement. Because we (and you) will want to test how your program plays without playing a complete game every time, we will generalize the game so that you can play with fewer (or more) holes and beans.

enum Side

You must have this enumerated type and constants of that type declared this way:

```
enum Side { NORTH, SOUTH };
```

Constants

You must have these constants declared this way:

```
const int NSIDES = 2;
const int POT = 0;
```

class Board

The Board class is responsible for maintaining the board. A Board object knows about the holes, pots, and beans. For those functions that talk about specific holes, the holes are indicated by a Side and hole number. The holes are numbered as indicated in the diagrams above; a pot is considered hole #0. Thus, North's hole #1 is next to North's pot; South's hole #6 is next to South's pot. **Notice that this numbering of holes is an aspect of the class's interface and our test code will depend on it.** Of course, your implementations of the functions are free to map that numbering scheme into something more convenient for their internal use.

The Board class must support these public member functions:

`Board(int nHoles, int nInitialBeansPerHole);`
Construct a Board with the indicated number of holes per side (not counting the pot) and initial number of beans per hole. If nHoles is not positive, act as if it were 1; if nInitialBeansPerHole is negative, act as if it were 0.

`int holes() const;`
Return the number of holes on a side (not counting the pot).

`int beans(Side s, int hole) const;`
Return the number of beans in the indicated hole or pot, or −1 if the hole number is invalid.

`int beansInPlay(Side s) const;`
Return the total number of beans in all the holes on the indicated side, not counting the beans in the pot.

`int totalBeans() const;`
Return the total number of beans in the game, including any in the pots.

`bool sow(Side s, int hole, Side& endSide, int& endHole);`
If the hole indicated by (s,hole) is empty or invalid or a pot, this function returns false without changing anything. Otherwise, it will return true after sowing the beans: the beans are removed from hole (s,hole) and sown counterclockwise, including s's pot if encountered, but skipping s's opponent's pot. The parameters endSide and endHole are set to the side and hole where the last bean was placed. (This function does *not* make captures or multiple turns; different Kalah variants have different rules about these issues, so dealing with them should not be the responsibility of the Board class.)

`bool moveToPot(Side s, int hole, Side potOwner);`
If the indicated hole is invalid or a pot, return false without changing anything. Otherwise, move all the beans in hole (s,hole) into the pot belonging to potOwner and return true.

`bool setBeans(Side s, int hole, int beans);`
If the indicated hole is invalid or beans is negative, this function returns false without changing anything. Otherwise, it will return true after setting the number of beans in the indicated hole or pot to the value of the third parameter. (This may change what beansInPlay and totalBeans return if they are called later.) This function exists solely so that we and you can more easily test your program: None of your code that implements the member functions of any class is allowed to call this function directly or indirectly. (We'll show an example of its use below.)

class Player

Player is an abstract base class that defines a common interface that all kinds of players (human and various computer players) must implement. It must support these public member functions:

`Player(std::string name);`
Create a Player with the indicated name.

`std::string name() const;`
Return the name of the player.

`virtual bool isInteractive() const;`
Return false if the player is a computer player. Return true if the player is human. Most kinds of players will be computer players.

`virtual int chooseMove(const Board& b, Side s) const = 0;`

Every concrete class derived from this class must implement this function so that if the player were to be playing side s and had to make a move given board b, the function returns the move the player would choose. If no move is possible, return −1.

`virtual ~Player();`

Since this class is designed as a base class, it should have a virtual destructor.

Each concrete class derived from Player will implement the chooseMove function in its own way. Of the classes listed here, only HumanPlayer::isInteractive should return true. (When testing, we may supply other kinds of interactive players.) Each of the three classes listed here must have a constructor taking a string representing the name of the player.

class HumanPlayer (derived from Player)

A HumanPlayer chooses its move by prompting a person running the program for a move (reprompting if necessary until the person enters a valid hole number), and returning that choice. We will never test for a situation where the user doesn't enter an integer when prompted for a hole number. (The techniques for dealing with the issue completely correctly are a distraction to this project, and involve either a function like `stoi` or `strtol`, or the type `istringstream`.)

class BadPlayer (derived from Player)

A BadPlayer is a computer player that chooses an arbitrary valid move and returns that choice. "Arbitrary" can be what you like: leftmost, nearest to pot, fewest beans, random, etc.. The point of this class is to have an easy-to-implement class that at least plays legally.

class SmartPlayer (derived from Player)

Here's your chance to shine. A SmartPlayer chooses a valid move and returns it. For any game played on a board of up to six holes per side, with up to four initial beans per hole, SmartPlayer::chooseMove must return its choice in no more than five seconds. (We'll give you a way of determining the time soon; until then, you can meet this requirement by limiting the depth of your game tree search or the number of game tree positions you explore to a limit you determine experimentally.) SmartPlayer::chooseMove will be worth about 15-20% of the points for this project.

class Game

This class manages a game of a particular size by configuring the board and playing the game. It must support these member functions:

`Game(const Board& b, Player* south, Player* north);`

Construct a Game to be played with the indicated players on a copy of the board b. The player on the south side always moves first.

`void display() const;`

Display the game's board in a manner of your choosing, provided you show the names of the players and a reasonable representation of the state of the board.

`void status(bool& over, bool& hasWinner, Side& winner) const;`

If the game isn't over (i.e., more moves are possible), set over to false and do not change anything else. Otherwise, set over to true and hasWinner to true if the game has a winner, or false if it resulted in a tie. If hasWinner is set to false, leave winner unchanged; otherwise, set it to the winning side.

`bool move();`

If the game is over, return false. Otherwise, make a complete move for the player whose turn it is (so that it becomes the other player's turn) and return true. "Complete" means that the player sows the seeds from a hole and takes any additional turns required or completes a capture. If the player gets an additional turn, you should display the board so someone looking at the screen can follow what's happening.

`void play();`

Play the game. Display the progress of the game in a manner of your choosing, provided that someone looking at the screen can follow what's happening. If neither player is interactive, then to keep the display from quickly scrolling through the whole game, it would be reasonable periodically to prompt the viewer to press ENTER to continue and not proceed until ENTER is pressed. (The ignore function for input streams is useful here.) Announce the winner at the end of the game. You can apportion to your liking the responsibility for displaying the board between this function and the move function. (Note: If when this function is called, South has no beans in play, so can't make the first move, sweep any beans on the North side into North's pot and act as if the game is thus over.)

`int beans(Side s, int hole) const;`

Return the number of beans in the indicated hole or pot of the game's board, or −1 if the hole number is invalid. This function exists so that we and you can more easily test your program.

This sample executable (for Windows, Mac, and Linux) was produced from this main routine:

```
int main()
{
        HumanPlayer hp("Marge");
        BadPlayer bp("Homer");
        Board b(3, 2);
        Game g(b, &hp, &bp);
```

```
        g.play();
}
```

whereas this one (for Windows, Mac, and Linux) is played between two computer players:

```
int main()
{
        BadPlayer bp1("Bart");
        BadPlayer bp2("Homer");
        Board b(3, 2);
        Game g(b, &bp1, &bp2);
        g.play();
}
```

Organizing your source files

So that we can test your program in ways that allow us to give you partial credit, there are oodles of requirements you must satisfy. Most of them are simple matters of code organization designed to prevent certain annoying dependencies.

Instead of having one class per file, for our ease of testing, you must organize your source code in the following manner:

Board.h
: Place your Board class definition here.

Board.cpp
: Place your Board member function implementations here.

Player.h
: Place your Player class definition here, as well as the definitions for the HumanPlayer, BadPlayer, and SmartPlayer classes.

Player.cpp
: Place your Player member function implementations here, as well as those for HumanPlayer, BadPlayer, and SmartPlayer.

Game.h
: Place your Game class definition here.

Game.cpp
: Place your Game member function implementations here.

Side.h
: This file must contain

```
enum Side { NORTH, SOUTH };

const int NSIDES = 2;
const int POT = 0;

inline
Side opponent(Side s)
{
    return Side(NSIDES - 1 - s);
}
```

It may contain additional types, constants, and non-member function declarations you find useful to add.

Main.cpp
: This file contains your main routine and any implementations of any additional non-member functions you find useful to add.

As always, header files must have appropriate include guards. If you wish, you may implement member functions with simple implementations directly in the class definition instead of the corresponding .cpp file. You must turn in all eight of these source files, even if some are empty (which could happen if you don't finish).

You must not add, remove, or change the declaration of any public member functions of the required classes (Board, Player and its subclasses, Game). (Exception: if the compiler-generated bookkeeping functions (destructor, copy constructor, and assignment operator) don't do the right thing, declare and implement your own.) You may add any *private* data members and *private* member functions you like to those classes. You may add protected member functions to those classes if you wish, but you must not add any protected data members. (We haven't talked about protected members in this course.) The word `friend` must not appear in your program.

If you want to add publicly accessible functionality to a class, do it through non-member functions declared in Side.h and implemented in Main.cpp. For example, given a board, both Game::move and SmartPlayer::chooseMove may want to make a complete move on that board for a player. You could have them both call a non-member function that takes a Board& parameter and uses only the public interface of Board.

If we create a project consisting of the eight required files, it must build successfully.

If we create a project consisting of the eight required files, but replace your Board.h and Board.cpp with our correct implementations that

behave as specified, the project must build successfully. Notice that this implies that the rest of your project must not depend on anything being in those Board files except what we said must be there. For example, if a Game member function depends on there being some additional non-member function relating to Boards, that function can't be in Board.h or Board.cpp. (While stylistically it should be, for our testing purposes you'll have to put such a function in Main.cpp and its declaration in Side.h.)

The provisions of the preceding paragraph similarly apply if we replace your Player.h and Player.cpp with ours. It also applies if we replace your Game.h and Game.cpp with ours. (Our replacement implementations of Board, Player and its subclasses, and Game all have correct copy and assignment behavior.)

If we replace your Board files with ours as indicated above, or your Player files, or your Game files, your functions must still behave correctly. This implies that you can't, for example, have a global variable that a Board member function looks at under the assumption that a Game member function set it a certain way. In other words, the only communication between your objects must be through the defined interfaces.

No member function of the required classes may cause anything to be written to cout except Game::display, Game::move, Game::play, and the chooseMove function of any class derived from Player for which isInteractive returns true. If you want to print things out for debugging purposes, write to cerr instead of cout. When we test your program, we will cause everything written to cerr to be discarded; we will never see that output, so you may leave those debugging output statements in your program if you wish.

No member function of the required classes may cause anything to be read from cin except Game::play and the chooseMove function of any class derived from Player for which isInteractive returns true.

As long as your program meets these requirements your main routine can do whatever you want, even nothing; whenever we run your program, we'll rename your main routine to something harmless and never call it. We'll append to your Main.cpp a main routine of our own.

During execution, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.

If we rename your main routine to something harmless and append the following to your Main.cpp, your program must build successfully. When the resulting executable is run, it must write `Passed all tests` and nothing more to cout and terminate normally.

```
#include "Board.h"
#include "Side.h"
#include <iostream>
#include <cassert>
using namespace std;

void doBoardTests()
{
    Board b(3, 2);
    assert(b.holes() == 3  &&  b.totalBeans() == 12  &&
                b.beans(SOUTH, POT) == 0  &&  b.beansInPlay(SOUTH) == 6);
    b.setBeans(SOUTH, 1, 1);
    b.moveToPot(SOUTH, 2, SOUTH);
    assert(b.totalBeans() == 11  &&  b.beans(SOUTH, 1) == 1  &&
                b.beans(SOUTH, 2) == 0  &&  b.beans(SOUTH, POT) == 2  &&
                b.beansInPlay(SOUTH) == 3);
    Side es;
    int eh;
    b.sow(SOUTH, 3, es, eh);
    assert(es == NORTH  &&  eh == 3  &&  b.beans(SOUTH, 3) == 0  &&
                b.beans(NORTH, 3) == 3  &&  b.beans(SOUTH, POT) == 3  &&
                b.beansInPlay(SOUTH) == 1  &&  b.beansInPlay(NORTH) == 7);
}

int main()
{
    doBoardTests();
    cout << "Passed all tests" << endl;
}
```

If we rename your main routine to something harmless and append the following to your Main.cpp, your program must build successfully. When the resulting executable is run, it must write two rows of equal signs and `Passed all tests`, and terminate normally. Nothing more may be written to cout except that between the two rows of equal signs something will be written to get Marge's move.

```
#include "Player.h"
#include "Board.h"
#include "Side.h"
#include <iostream>
#include <cassert>
using namespace std;

void doPlayerTests()
```

```
{
    HumanPlayer hp("Marge");
    assert(hp.name() == "Marge"  &&  hp.isInteractive());
    BadPlayer bp("Homer");
    assert(bp.name() == "Homer"  &&  !bp.isInteractive());
    SmartPlayer sp("Lisa");
    assert(sp.name() == "Lisa"  &&  !sp.isInteractive());
    Board b(3, 2);
    b.setBeans(SOUTH, 2, 0);
    cout << "=========" << endl;
    int n = hp.chooseMove(b, SOUTH);
    cout << "=========" << endl;
    assert(n == 1  ||  n == 3);
    n = bp.chooseMove(b, SOUTH);
    assert(n == 1  ||  n == 3);
    n = sp.chooseMove(b, SOUTH);
    assert(n == 1  ||  n == 3);
}

int main()
{
    doPlayerTests();
    cout << "Passed all tests" << endl;
}
```

If we rename your main routine to something harmless and append the following to your Main.cpp, your program must build successfully. When the resulting executable is run, it must terminate normally. The last line written to cout must be `Passed all tests`.

```
#include "Game.h"
#include "Player.h"
#include "Board.h"
#include "Side.h"
#include <iostream>
#include <cassert>
using namespace std;

void doGameTests()
{
    BadPlayer bp1("Bart");
    BadPlayer bp2("Homer");
    Board b(3, 0);
    b.setBeans(SOUTH, 1, 2);
    b.setBeans(NORTH, 2, 1);
    b.setBeans(NORTH, 3, 2);
    Game g(b, &bp1, &bp2);
    bool over;
    bool hasWinner;
    Side winner;
      //    Homer
      //   0  1  2
      // 0         0
      //   2  0  0
      //    Bart
    g.status(over, hasWinner, winner);
    assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 &&
        g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 2 &&
        g.beans(SOUTH, 1) == 2 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

    g.move();
      //   0  1  0
      // 0         3
      //   0  1  0
    g.status(over, hasWinner, winner);
    assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&
        g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 0 &&
        g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

    g.move();
      //   1  0  0
      // 0         3
      //   0  1  0
    g.status(over, hasWinner, winner);
    assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&
        g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&
        g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

    g.move();
```

```
      //   1  0  0
      // 0         3
      //   0  0  1
    g.status(over, hasWinner, winner);
    assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&
        g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&
        g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 1);

    g.move();
      //   0  0  0
      // 1         4
      //   0  0  0
    g.status(over, hasWinner, winner);
    assert(over && g.beans(NORTH, POT) == 1 && g.beans(SOUTH, POT) == 4 &&
        g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&
        g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);
    assert(hasWinner && winner == SOUTH);
}

int main()
{
    doGameTests();
    cout << "Passed all tests" << endl;
}
```

Turn it in

By May 20, there will be a link on the class webpage that will enable you to turn in your source files and report. You will turn in a zip file containing these nine files:

- Eight C++ files containing your program. These are the files listed above. Comment any function you add to indicate what it does. Comment any non-trivial code.

- report.docx, report.doc, or report.txt, a report (either a Word document or a text file) containing

    - a description of the design of your classes. We know what the public interfaces are, but what about your implementations: What are the major data structures that you use? What private member functions or helper non-member functions did you define for what purpose?

    - a description of your design for SmartPlayer::chooseMove, including what heuristics you used to evaluate board positions.

    - pseudocode for non-trivial algorithms.

    - a note about any known bugs, serious inefficiencies, or notable problems you had.

    - a list of the test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. Even if you do not correctly implement all the functions, you can still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I *had* implemented it."

Advice

Set up the eight files first, with stub implementations of the functions; for example, have functions returning an int be implemented as

```
return -9999;  // TODO:  implement this function
```

Get this to build successfully. Then implement the Board functionality and test it. Implement and test Player/BadPlayer next, then HumanPlayer, then Game. If you then make SmartPlayer choose a move the same way BadPlayer does, you have a working program that will be worth a majority of the correctness points. (Ours at this stage is about 315 lines of code, counting only lines that have at least one non-punctuation character outside of a comment.)

After this works perfectly, try to earn the rest of the correctness points by making SmartPlayer choose its moves intelligently. No matter how intelligently you try to make SmartPlayer play, it won't earn any of these points if it ever chooses an illegal move.

When you're sure SmartPlayer::chooseMove is correct, to test it for speed, build an optimized version of it. On cs32.seas.ucla.edu, build the executable and run it this way:

```
g32fast -o kalah Main.cpp Board.cpp Player.cpp Game.cpp
./kalah
```

(You don't have to know this, but this command omits some of the runtime error checking compiler options that our g32 command supplies,

and it adds the -O2 compiler option that causes the compiler to spend more time optimizing the machine language translation of your code so that it will run faster when you execute it.)

Under Xcode, select Product / Scheme / Edit Scheme.... In the left panel, select Run, then in the right panel, select the Info tab. In the Build Configuration dropdown, select Release. For Visual C++, it's a little trickier.

Your program will run much faster in Release mode than Debug mode, but you won't be able to get much information from the debugger if you're trying to track down a bug. (You can always switch back from Release to Debug mode if you like.) When we test your program, we will build it in Release mode and/or build it with g32fast.