**BLACKJACK**

**A CARD GAME**

**Ireoluwa Dairo**

**CSC/CIS 17C -42513**

**SPRING 2025**

**Project 2**

**INTRODUCTION**

This project is a modified version of a C++ Blackjack game that makes use of the Standard Template Library (STL), trees, recursions, recursive sort and hashing. I chose Blackjack because it's a well-known card game with clear rules. The game follows classic Blackjack rules: get as close to 21 as possible without going over.

At the start of each round, both player and dealer receive two cards. Players can "hit" (draw cards), "stand" (keep hand), "double down" (double bet for one card), or request strategy "hints" from an advisor. The dealer follows standard casino rules, hitting until reaching 17 or higher. The game determines winners by comparing final totals without busting.

Github - [CSC\_17C/project at main · Kvngjaid04/CSC\_17C](https://github.com/Kvngjaid04/CSC_17C/tree/main/project)

**Summary**

This project has over 1200 lines of code and 40+ functions, using f several C++ concepts:

* **STL Containers:** vector, list, map, set, stack, and queue
* **STL Algorithms:** for\_each, transform, count\_if, generate, random\_shuffle, find
* **Iterators:** used throughout the code
* **Lambda functions:** paired with STL algorithms for cleaner logic
* **Binary Tree Operations:** Tree rotations, balancing, and strategy lookup algorithms
* **Recursive Functions:** Player turn logic with depth tracking, double-down validation
* **Recursive Sorting:** Merge sort implementation for deck display functionality
* **Hash Functions:** djb2 primary hash with secondary hash for collision resolution
* **Tree Algorithms:** AVL rotations (LL, RR, LR, RL) for maintaining balance

**Hash Table Implementation:** The custom Hash class uses double hashing to store player profiles. When players log in, the hash table quickly retrieves their data using the djb2 hash function with collision resolution. This allows multiple players to save/load their progress efficiently.

**AVL Tree for Strategy (hints):** The AVL tree stores Blackjack strategy based on player total and dealer upcard.

**Recursive Design:** Player turns use recursive function calls with depth parameters. The double-down feature uses recursion with depth tracking to validate betting logic. Merge sort recursively sorts the deck for display purposes.

**Why Graphs Weren't Used:** I couldn't find or think of a logical application within Blackjack's framework. The game is more about randomness and probability rather than connected relationships or pathfinding.

**Description**

The game features a player management system that uses hash tables for quick data retrieval and storage. The AVL tree serves as a strategy advisor, providing optimal move recommendations based on current hand values and dealer cards.

New players begin with $1000 starting cash and each round follows a structured flow: players place bets with stack-based history tracking, receive cards stored in vector containers, and make strategic decisions guided by the advisor. The recursive player turn system manages complex game states . After each game, players can view statistics including card frequency analysis through maps, unique card tracking via sets, and complete action history maintained in queues.

Player progress persists through file operations integrated with hash table management for efficient data handling. The merge sort algorithm provides organized deck viewing capabilities. Gameplay continues until players choose to quit or deplete their funds.

**Sample Input/Output**

A computer screen with text and numbers

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

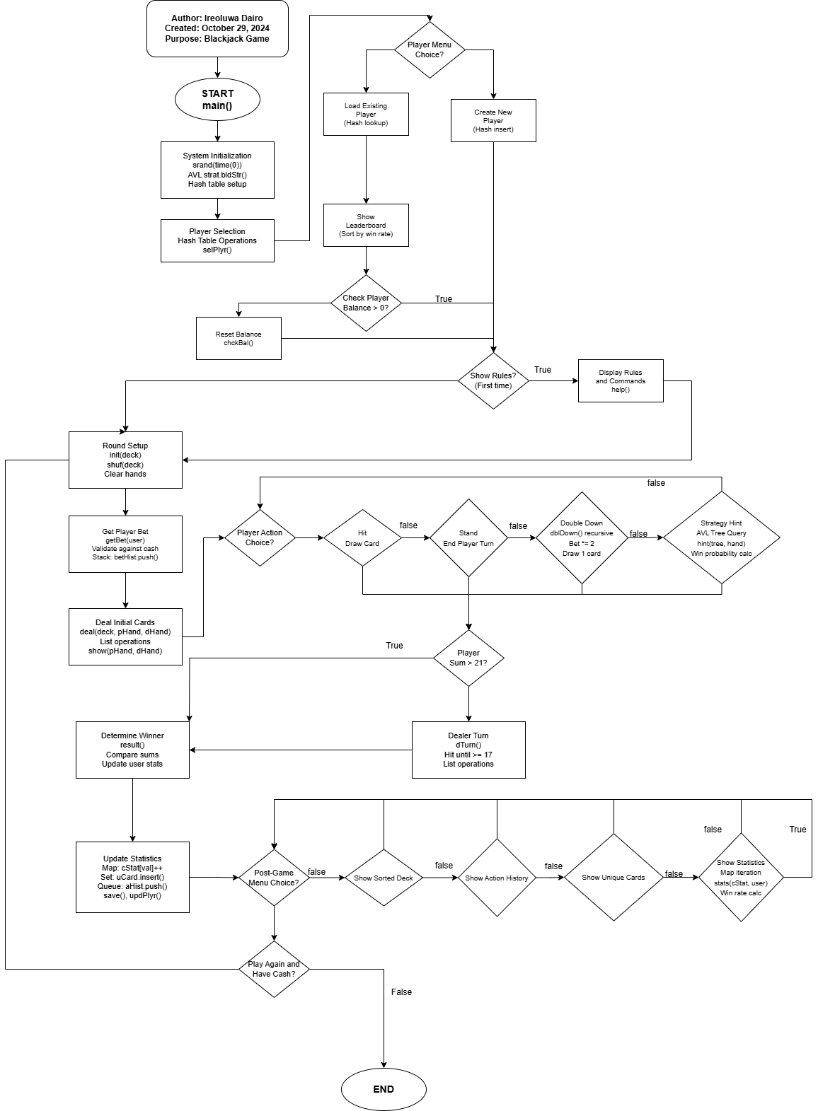
AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**FLOWCHART**

Note: This flowchart represents the general logic of the code, not the exact execution flow.

****

**Pseudocode**

Start program

Initialize random number generator using srand(time(0))

Create STL containers:

- vector<Card> deck for the card deck

- list<Card> phand, dhand for player and dealer hands

- map<int, int> cstat for card statistics

- set<Card> ucard for tracking unique cards

- queue<string> ahist for recording game actions

- stack<int> bethist for betting history

- Hash allplyr for player data storage

- AVL strat for strategy hints

Initialize AVL tree with blackjack strategy using strat.bldstr()

Load existing player files into hash table using lodexst()

Call selplyr() function to select or create player using hash operations

Repeat the game loop:

- Initialize the deck using init(deck, ncards) with vector operations

- Shuffle deck using shuf(deck) with STL random\_shuffle algorithm

- Ask for rules display if first time using askrul()

- Get the player's bet using getbet() and validate against user.cash

- Push bet to betting history stack using bethist.push(bet)

- Clear player and dealer hands using phand.clear(), dhand.clear()

- Deal initial cards using deal() function, adding to lists with push\_back()

- Display the initial cards using show() function

- Record "Initial deal" action in history queue using ahist.push()

Play the player's turn using pturn() recursively:

- Calculate hand value using sumhand() with STL for\_each algorithm

- Display current total and options (hit/stand/double/hint)

- If hit requested, draw card and add to player's list using push\_back()

- If double requested, call dbldown() recursive function with depth parameter

- If hint requested, query AVL tree using strat.gethint() for strategy advice

- Update hand sum and check for bust condition

- Continue recursively until stand or bust

Record "Player turn complete" in action history using ahist.push()

If player has not busted (psum <= 21), play dealer's turn:

- Call dturn() function using list iterators

- Hit until dealer sum >= 17 using while loop

- Display dealer cards using list iteration

Record "Dealer turn complete" in action history using ahist.push()

Handle the outcome using result() function:

- Compare final totals

- Update user.wins, user.games, user.cash

- Display winner and cash remaining

Save the game state using save(user, fname) to binary file

Update player in hash table using updplyr(user)

Update card statistics using STL containers:

- Iterate through player hand with range-based for loop

- Update map using cstat[val]++ for each card value

- Insert cards into set using ucard.insert() for unique tracking

Display post-game menu using getmenu():

- Show card statistics using stats() with map iteration and for\_each

- Report unique cards count from set container using ucard.size()

- Display action history from queue using temporary queue and front()/pop()

- Show sorted deck using merge sort algorithm srtdeck() and mrg()

- Allow player to choose next action

Check if player can continue using canplay():

- Verify user.cash > 0

- If broke, display game over message

If playing again:

- Clear all containers using reset() function

- Clear hands with phand.clear(), dhand.clear()

- Empty action history queue using while(!Ahist.empty()) ahist.pop()

- Reset deck size to SIZE constant

- Continue to next iteration

End game loop when player quits or runs out of money

Display final goodbye message

Clean up all STL containers automatically via destructors

End program

|  |
| --- |
| Card |
| -suit: char  -Val: int |
| + operator<()  +operator ==() |

|  |
| --- |
| save |
| -name: char[30]  -wins: int  -games: int  -cash: int |
| +operator<() |

|  |
| --- |
| <<enumeration>>  Suit |
| HEART  DIAM  SPADE  CLUB |

**UML**

|  |
| --- |
| AVL |
| +root : Node\* |
| +AVL()  +~AVL()  +height(Node\*)  +diff(Node\*)  +llRot(Node\*)  +rrRot(Node\*)  +lrRot(Node\*)  +rlRot(Node\*)  +bal(Node\*)  +ins(Node\*,int,char)  +ins(int, char)  +find(Node\*, int)  +getHint(int, int)  +bldStr()  +clear(Node\*)  +show(Node\*, int)  +inOrd(Node\*) |

|  |
| --- |
| Hash |
| -tbl : vector<PlyrEnt>  -tblSz : int  -numUsr : int |
| +Hash(int sz=13)  +add(string, Save)  +get(string)  +getAll()  +empty()  +count()  -hash1(string)  -hash2(string) |

|  |
| --- |
| <<struct>>  Node |
| +data : int  +hint : char  +left : Node\*  +right : Node\* |
| +Node() |

|  |
| --- |
| <<struct>>  PlyrEnt |
| +name : string  +data : Save  +actv : bool |
| +PlyrEnt()  +PlyrEnt(string,Save) |

|  |
| --- |
| **Menu.h/cpp** |
| **+help()**  **+again()**  **+stats(map<int,int>&, Save&)**  **+getMenu(map<int,int>&, set<Card>&, queue<string>&, Save&)** |

|  |
| --- |
| Play.h/cpp |
| +sumHand(list<Card>&)  +dblDown(Save&, int&,  int=0)  +deal(vector<Card>&,  list<Card>&,list<Card>&)  +show(list<Card>&, list<Card>&, Vector<Card>&)  +hint(AVL&, list<Card>&,  int, int)  +pTurn(vector<Card>&, list<Card>&, int&, Save&, int&, AVL&)  +dTurn(vector<Card>&, list<Card>&, int&)  +result(Save&, int, int, int)  +canPlay(Save&) |

|  |
| --- |
| Usermgr.h/cpp |
| +load(Save&, string)  +save(Save, string)  +valname(string)  +clrbuf()  +getbet(Save&)  +calcrte(int, int)  +lodexst()  +valdata(Save&)  +selplyr()  +updplyr(Save&)  +lodexst(vector< plyrent>&)  +creatnw()  +shwlead(vector< plyrent>&)  +rsetbal(vector< plyrent>&)  +chckbal(Save&) |

|  |
| --- |
| Deck.h/cpp |
| +init(vector<Card>&, int&)  +shuf(vector<Card>&)  +draw(vector<Card>&)  +getSuit(int)  +getVal(int)  +pCard(int, char)  +srtDeck(vector<Card>&,int, int)  +mrg(vector<Card>&, int, int, int)  +showDck(vector<Card>&) |

|  |
| --- |
| STL Containers in main() |
| Vector<Card> deck(SIZE) - Card deck storage  List<Card> phand, dhand - Player/dealer hands  Map<int, int> cstat - Card statistics tracking  Set<Card> ucard - Unique cards drawn  Queue<string> ahist - Action history log  Stack<int> bethist - Betting history stack |

**CHECKOFF SHEET**

1. Container classes (Where in code did you put these Concepts and how were they used?
2. Sequences (At least 1)
   1. **List:**

**main.cpp (ln 49-50) - list<Card> pHand; list<Card> dHand; store player/dealer cards**

Play.cpp (ln 63,72) - pHand.push\_back(drawn); - add cards to hand

* 1. slist
  2. bit\_vector:

main.cpp (ln 48)- vector<Card> deck(SIZE); - main deck storage

Deck.cpp (ln 23) - deck.push\_back(c); - build deck

Deck.cpp (ln 38) - deck.pop\_back(); - draw cards

1. Associative Containers (At least 2)
2. **Set:**

main.cpp (ln58) - set<Card> uCard; - track unique cards drawn

main.cpp (ln 161) - uCard.insert(\*it); - add unique cards

1. **Map:**

main.cpp (ln 57) - map<int, int> cStat; - card statistics counter

main.cpp (ln 160) - cStat[val]++; - increment card count

Deck.cpp (ln 54) - map<int, string> vluNmes - card name lookup

1. Hash
2. Container adaptors (At least 2)
3. **Stack:**

main.cpp (ln 60) - stack<int> betHist; - betting history

main.cpp (ln 132) - betHist.push(bet); - record bets

1. **Queue:**

main.cpp (ln 59) - queue<string> aHist; - action history

main.cpp (ln 142) - aHist.push("Initial deal"); - log game actions

1. priority\_queue
2. Iterators
3. Iterators
   1. Concepts (Describe the iterators utilized for each Container)
4. Trivial Iterator
5. Input Iterator
6. Output Iterator
7. Forward Iterator:

Menu.cpp (ln 79) - stat.begin(), stat.end() - traverse map for statistics

1. Bidirectional Iterator:

Play.cpp (ln 179) - for (auto it = hand.begin(); it != hand.end(); ++it) - traverse card lists

1. Random Access Iterator:

Deck.cpp (ln 76) - deck.begin() + lft - direct position access for merge sort

1. Algorithms (Choose at least 1 from each category)
2. Non-mutating algorithms
3. **for\_each:**

Play.cpp (ln 37) - for\_each(hand.begin(), hand.end(), [&](const Card& card) - calculate hand sum

Menu.cpp (ln 79) - for\_each(stat.begin(), stat.end(), [](const pair<int, int>& item) - display stats

1. find
2. count:

UserMgr.cpp (ln 50) - count\_if(name.begin(), name.end(), [](char c) - validate username chars

1. equal
2. search
3. Mutating algorithms
4. copy
5. Swap
6. Transform:

Menu.cpp (ln 110) - transform(input.begin(), input.end(), input.begin(), ::tolower); - convert to lowercase

1. Replace
2. fill
3. Remove
4. Random\_Shuffle:

Deck.cpp (ln 31) - random\_shuffle(deck.begin(), deck.end()); - shuffle deck

1. Organization
2. Sort:

UserMgr.cpp (ln 208) - sort(players.begin(), players.end(), [](const PlyrEnt& a, const PlyrEnt& b) - sort leaderboard

1. Binary search
2. Merge:

Deck.cpp (ln 96) - srtDeck() - custom merge sort implementation

1. inplace\_merge
2. Minimum and maximu

**Variables and Data Structures**

**STL Containers:**

**main.cpp (ln 48-50,57-60) - deck, hands, stats, history containers**

Structures with Custom Operator:

Card.h (ln 17,24) - operator< and operator== for set/map usage

Save.h (ln 21) - operator< for player comparison

**Concepts used**

* Containers: vector, list, set, map, stack, queue
* Algorithms: for\_each, count\_if, transform, random\_shuffle, sort
* Iterators: bidirectional (list), random access (vector), forward (set/map)
* Lambda expressions: hand calculation, validation, sorting, display

**REFERENCES**

1. Originally developed for CSC-5, enhanced for CSC-17A, and now updated with STL implementation, hashing, trees, recessions and recursive sort for CSC-17C by Ireoluwa Dairo
2. C++ STL Documentation [Standard Template Library Programmer's Guide](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/)
3. Blackjack rules: <https://www.bicyclecards.com/how-to-play/blackjack/>