Segment Tree Project Report

Brandon Charette, Andrew Dionizio, Kaidan Campbell, Kyle DaSilva

University of Rhode Island

CSC 212: Data Structures and Abstractions

Jonathan Schrader

November 29, 2022

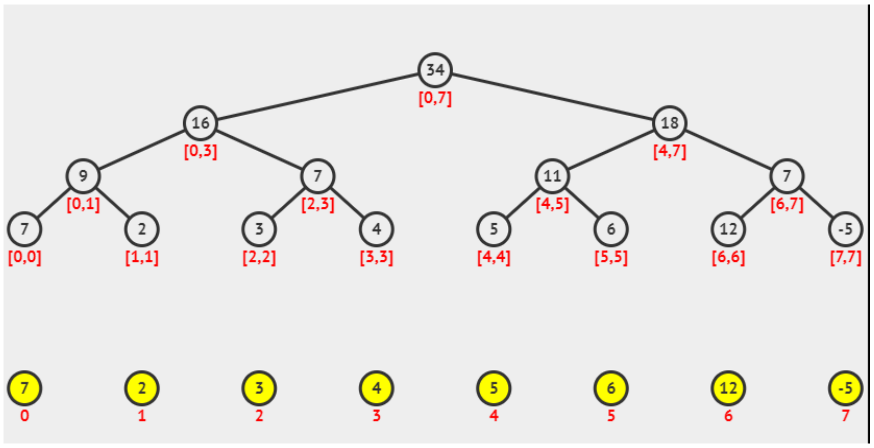
Abstract

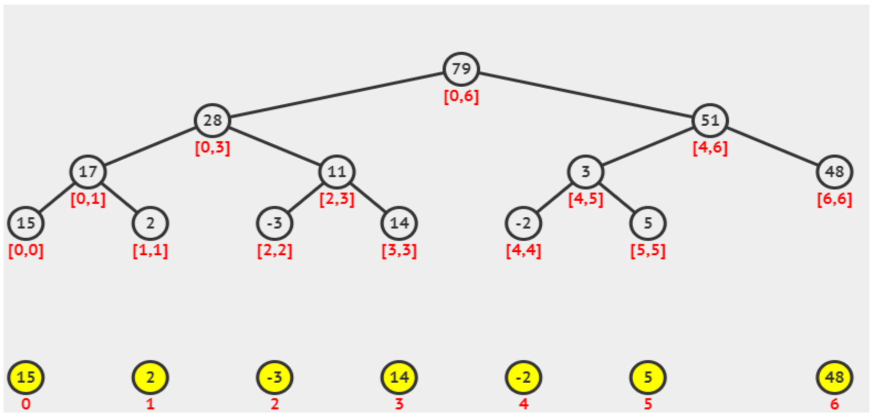
The data structure of a segment tree will be implemented to find different probabilities while a user is playing a game of blackjack against a user. Some of these probabilities are the odds they will exceed 21 on their next move, the odds of the player having a favorable hand on the next move, and the odds in which the dealer will have a better hand on the next move. The purpose of this program is to have the player have a better gauge of the game at hand, to help them to make better decisions while playing.

Introduction

A segment tree is a data structure that stores array intervals into a tree. In an array, the leaves of the tree would be all the elements in that array. In this tree, the nodes other than the leaves must have exactly two children. From there, the parent node of two leaves is the sum of the two leaves. The root of the tree is the total sum of all the elements in the array. A segment tree can be created in runtime of O(n log n).

While building a segment tree it essentially comes down to an explicit divide-and-conquer application. For instance, say we have an array *a* and size *n* of array *a*. The sum of the segment's index would be *a*[0:*n* -1]. From there, the array can be split into two halves denoting *a*[0:*n*/2 -1] and *a*[*n*/2: *n* -1]. This can be continued sequentially until all segments have been created. Segment trees work ideally for both arrays that contain an even and odd number of elements. Below is a segment tree is an array of even size containing the integers { 7 , 2 , 3 ,4 ,5 ,6 , 12 , -5} and an array with odd size elements with { 15, 2, -3, 14, -2, 5 ,48}.





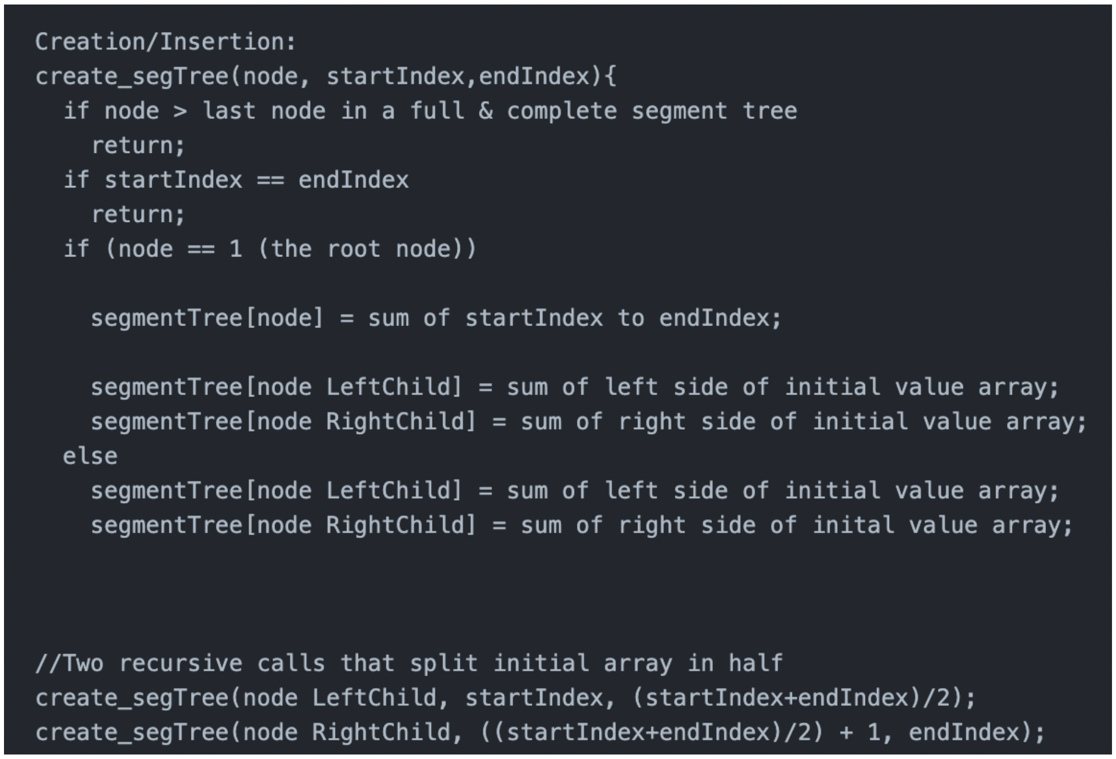
Segment trees have many different applicable uses. Finding the maximum, minimum, and finding different sum queries are all common applicants of this tree. Finding the sum over a certain interval index is another essential use for this data structure. In the two examples above, the intervals are shown in red. The user can just go up the tree to find what certain segment interval they want.

Blackjack is a popular card game found at casinos. The objective of blackjack is to be dealt cards with a higher count than the cards of the dealer without exceeding 21. When dealt, aces can count as 1 or 11, face cards count as 10, and the numbered cards count as the number shown on the card. Bets are usually placed before the cards are dealt but the rules of this can vary. After bets are placed, two cards are dealt to all the players and the dealer. You would be able to see both cards you have and one of the two cards that the dealer has. With this information, you can either decide to hit or stand. If you decided to hit then the dealer would deal you another card which would add to the count that you currently have. If you decided to stand then your final count would be the count you have in your hand. Once you decide to stand the dealer will deal cards to themself until they get to at least a count of 17 or they bust. When saying bust this means that the count is exceeding 21 and would automatically make whoever just busted lose.

Implementation

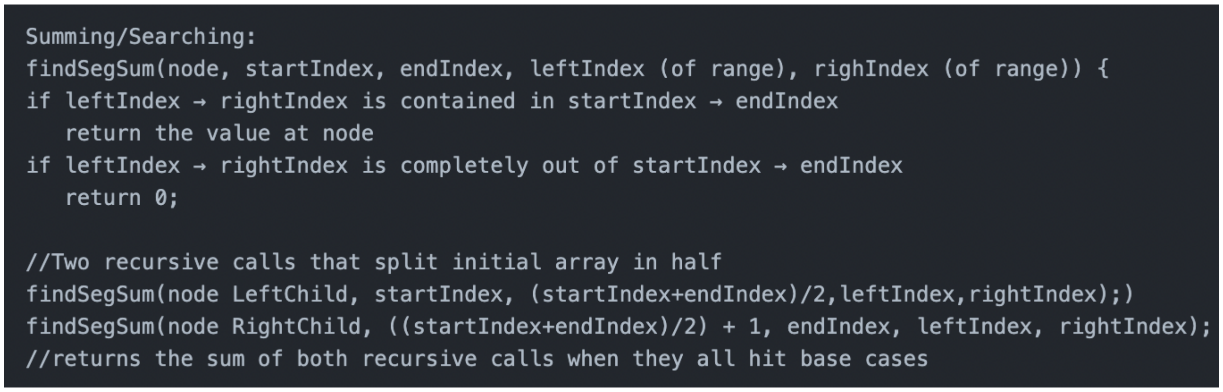
Our group decided to determine different types of probabilities for a player's hand during a blackjack hand. While planning out the project, a blackjack and a segment tree program must be created. The segment was a class created into two files: SegTree.cpp and SegTree.h. The blackjack program and implementation of the probabilities were created in a main.cpp file.

A class object was used to create a segment tree in SegTree.cpp. This allows for segment trees to be used for programs other than this project. While creating a specific tree that was asked by a user, the root is the sum of all the elements in the array and each branch was created recursively. The pseudocode for the insertion method of the tree is provided below:



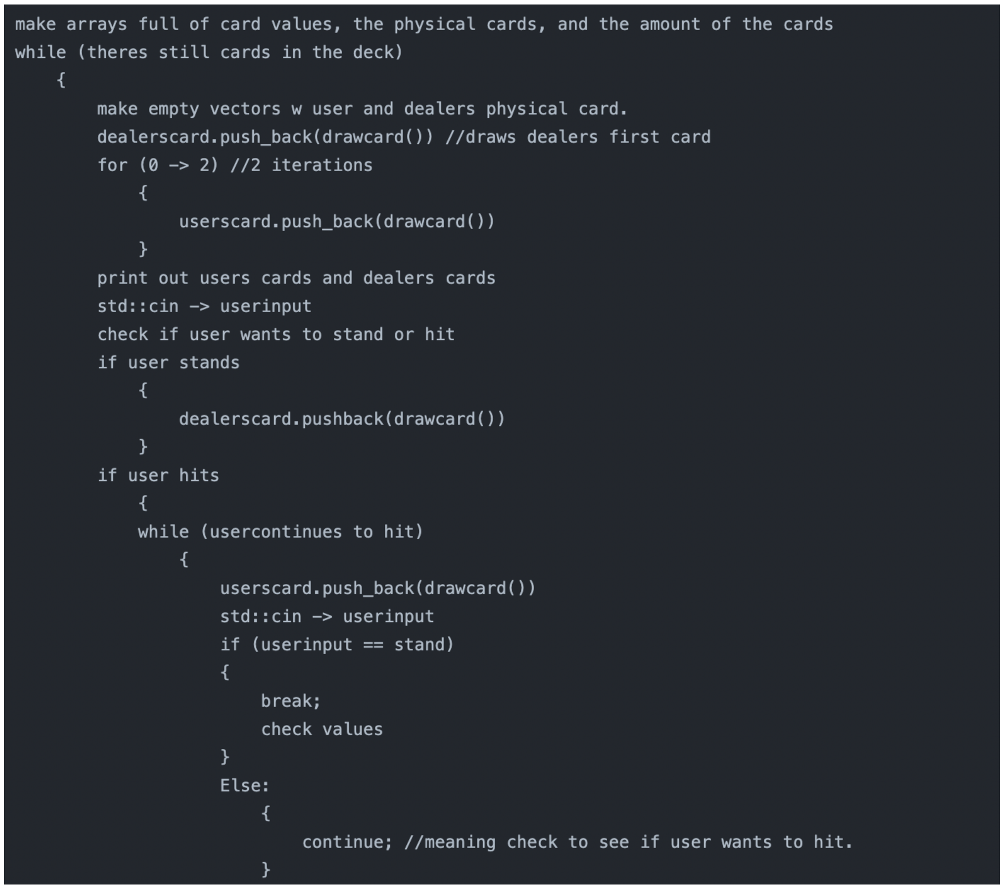
```
Creation/Insertion:
create_segTree(node, startIndex,endIndex){
  if node > last node in a full & complete segment tree
    return;
  if startIndex == endIndex
    return;
  if (node == 1 (the root node))

    segmentTree[node] = sum of startIndex to endIndex;

    segmentTree[node LeftChild] = sum of left side of initial value array;
    segmentTree[node RightChild] = sum of right side of initial value array;
  else
    segmentTree[node LeftChild] = sum of left side of initial value array;
    segmentTree[node RightChild] = sum of right side of inital value array;



//Two recursive calls that split initial array in half
create_segTree(node LeftChild, startIndex, (startIndex+endIndex)/2);
create_segTree(node RightChild, ((startIndex+endIndex)/2) + 1, endIndex);
```

Another functionality of the segment tree used throughout the program is the summing or searching method. This method was also created with a recursively calling function. This method was also included in the segment tree object. The pseudocode for this can also be seen below:



```
Summing/Searching:
findSegSum(node, startIndex, endIndex, leftIndex (of range), righIndex (of range)) {
if leftIndex → rightIndex is contained in startIndex → endIndex
   return the value at node
if leftIndex → rightIndex is completely out of startIndex → endIndex
   return 0;

//Two recursive calls that split initial array in half
findSegSum(node LeftChild, startIndex, (startIndex+endIndex)/2,leftIndex,rightIndex);)
findSegSum(node RightChild, ((startIndex+endIndex)/2) + 1, endIndex, leftIndex, rightIndex);
//returns the sum of both recursive calls when they all hit base cases
```

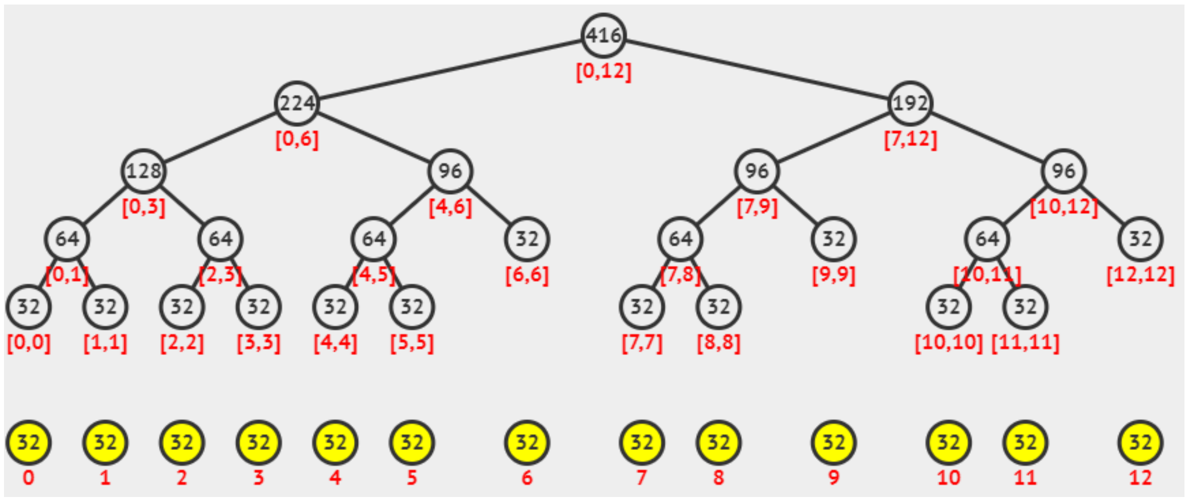
In the blackjack portion of the project, a program was created to play blackjack against a user. In this program, a total of eight decks are used, for a total of 416 possible cards in the game. The cards are then shuffled and dealt two cards to the dealer and two cards to the user. As always, the dealer is only showing one card. The program then tells the user their cards and the total value of their hand. From there, two options are given, hit or stand. A user can repeatedly hit until they either stand, or their score exceeds 21. The dealer will then play their turn, and will always stand if their hand is 17 or greater. The scores are then compared to see who won and allow the users to replay as many times as they like. Below is the pseudocode for blackjack:



```
make arrays full of card values, the physical cards, and the amount of the cards
while (theres still cards in the deck)
    {
        make empty vectors w user and dealers physical card.
        dealerscard.push_back(drawcard()) //draws dealers first card
        for (0 -> 2) //2 iterations
            {
                userscard.push_back(drawcard())
            }
        print out users cards and dealers cards
        std::cin -> userinput
        check if user wants to stand or hit
        if user stands
            {
                dealerscard.pushback(drawcard())
            }
        if user hits
            {
            while (usercontinues to hit)
                {
                    userscard.push_back(drawcard())
                    std::cin -> userinput
                    if (userinput == stand)
                    {
                        break;
                        check values
                    }
                    Else:
                        {
                            continue; //meaning check to see if user wants to hit.
                        }
```

To create this program different functions were created. Creating the deck, drawing a card, and comparing scores are all functions the code contains. Creating the deck creates an array with size 13 that contains all possible card values. This deck is then duplicated eight times to create the full deck. This deck is then shuffled by utilizing the “random” library. The drawing card function will draw cards. More importantly, though, this function keeps track of the total number of cards remaining and will refill a deck if the deck size is equal to zero. Lastly, the check scores use simple if statements to check the winning hand score.

The last part of our program is using the segment tree for different probabilities. The image below is a visualization of the segment tree being utilized to determine these probabilities.



This tree contains 13 leaves representing each possible value in a deck of cards. They are all at value 32 since there are four cards in a deck and eight decks in total. This tree being used is also a full and complete binary tree. Each of these probabilities would be new functions made in SegTree.cpp. Creating probability will give the player better analytical judgment on whether or not they should decide to hit or stand.

The first probability that can be utilized is the odds of the user’s next card not making their score exceed 21. The program will take the user's score and find the next card that would not make the score exceed 21 and then find that probability using the tree. For example, if a player has two cards a 7 and 6, making their hand value 13. From there the program will determine which card's values are valid by looping through the original array and determining if adding that number will not exceed 21. Based on this example cards { 2, 3,4, 5,6,7,8, Ace} are all valid. These values will then be traversed in the tree to determine the sum of all possible valid cards. By using the tree this would be 256 possible cards. However, the program would compute 254 possible cards since the user having a 7 and a 6 would remove that possibility from the total. This number of 254 would then be divided by the root of the tree. In this scenario, the program would have 413 possible cards. Dividing these two numbers would give a 61.05% chance of having a card that would not make your hand exceed 21.

Another probability created for this project is the percentage that a player will have a favorable hand. A favorable hand is a total score between 17 to 21. By using the same example in the previous probability the cards {4, 5,6,7,8, 9} are all valid cards. This would also be done with a loop to add the card to the total to check if the total is valid. Then by traversing the tree 190 possible cards out of 413 is what the function would compute. After dividing the two numbers a 46.00% chance of a favorable hand would exist.

A third probability found in this program is the chance that the dealer's next card will make their hand better than yours. This implementation was created with a separate function in our SegTree.cpp. This function would essentially first take a variable that would be the player's total score minus the one card that the dealer is showing. After having that variable it would then find cards in the original array that would make the dealer's score higher than the user's. The purpose of this probability is to help the player have a better understanding of the dealer's hand to decide if it could be the right time to stand.

Contributions

Throughout this month-long process, our group worked towards finishing this project. During this period, our group met almost every day to discuss what needed to be done and what progress was being made. Allocating time to meet with each other helped with the entire organization and communication of this project. Other than this report, the code, the GitHub repository, and a presentation were created.

Andrew Dionizio created the segment tree section of the report. Kaidan Campell had started the blackjack section and Brandon Charette and Kyle DaSilva had finished it and cleaned it up. Each of us had created different functions to determine each of the probabilities. This report was mostly made by Brandon with assistance from Kyle.

References

cplusplus.com. (n.d.). Retrieved December 4, 2022, from https://cplusplus.com/

Crick, J. (2018, May 1). *Practical data structures for frontend applications: When to use segment trees*. HackerNoon. Retrieved December 4, 2022, from https://hackernoon.com/practical-data-structures-for-frontend-applications-when-to-use-segment-trees-9c7cdb7c2819

Segment tree. Segment Tree - Algorithms for Competitive Programming. (2022, July 14). Retrieved December 4, 2022, from https://cp-algorithms.com/data_structures/segment_tree.html#structure-of-the-segment-tree

VisualGO. VisuAlgo - segment tree. (n.d.). Retrieved December 4, 2022, from http://cszqwe.github.io/VisuAlgo_SegmentTree/