CT077-3-2-DSTR

APU2F2402CS(AI)

HAND OUT DATE: **28th AUGUST 2024**

HAND IN DATE: **13th OCTOBER 2024**

WEIGHTAGE: **30%**

LECTURER NAME: **LEE KIM KEONG**

HOTEL REVIEWS SENTIMENT ANALYSIS WITH ARRAYS AND LINKED LISTS

PREPARED BY: GROUP J

TP Number	Student Name
ANGELINA LEANORE	TP072929
BRITTNEY LAU KO XUAN	TP065461
CHEAH MEI SUEN	TP064568
LEE JAMIE	TP066530

Table of Contents

1. Workload Matrix

Component	**Student 1 Name:** Angelina Leanore 	**Student 2 Name:** Brittney Lau Ko Xuan 	**Student 3 Name:** Cheah Mei Suen 	**Student 4 Name:** Lee Jamie 	**Total**
a) Code	25%	25%	25%	25%	**100%**
b) Documentation	25%	25%	25%	25%	**100%**

2. Design and Implementation

2.1 System Functions and Source code

2.1.1 Data Containers and Structures

In this section, the implementation of self-created data containers is shown. The data containers created are dynamic array, and doubly linked list. This section also shows the implementations of the Review and Word structure, used to create objects and allow encapsulation in our Object-orientated Programming (OOP) approach.

Data Container- Dynamic Array

```
template <typename T>
class DynamicArray
{
  T *dataArray;
  int capacity;
  int size;
```

Figure 1 Structure for the Dynamic Array Data Container

Above shows the structure of our dynamic array. It consists of three variables, *dataArray* of variable data type, and integers *capacity* and *size*. The *dataArray* variable stores the pointer to the actual static array which stores the data. Whereas *capacity* is used to initialize the size of the actual static array, and the variable *size* is used to keep track of the number of elements which have data appended into the actual static array.

Resize function

```
private:
  void resize()
  {
    capacity *= 2;
    T *newArray = new T[capacity]; // new array in the stack, gets deleted after function finish executing
    for (int i = 0; i < size; i++)
    {
      newArray[i] = dataArray[i]; // Copy elements
    }
    delete[] dataArray;   // Delete old array
    dataArray = newArray; // Point to the new array
  }
```

Figure 2 Resize() function of the Dynamic Array Data Container.

Above is the source code for the resize() function used in the Dynamic Array Data Container. This function is used when the size of the dynamic array has reached the maximum capacity of the array (size = capacity), meaning when all allocated memory of the static array is foiled up with data. The resize function first doubles the capacity for the

initialization of a new static array with double the memory allocation. Once a new static array with doubled capacity is initialized, elements in the original static array is copied into the new static array. Once all elements are copied successfully, the original and now referred to as the old static array is deleted. The *dataArray* pointer of the dynamic array points to the newly created static array, which now has doubled memory allocated to append new elements.

Data Container- Doubly Linked List

Node of the Doubly Linked List

```
template <typename T>
struct DoublyNode
{
    T data;
    DoublyNode<T> *next;
    DoublyNode<T> *prev;
};
```

Figure 3 Structure of the DoublyNode used in the Doubly Linked List

The above shows the structure of the node used in the created Doubly Linked List data container, DoublyNode. The DoublyNode consists of three variables- the data, the pointer to the next node, and a pointer to the previous node. The data stores the actual data of the node, and the *next* and *prev* pointers link all nodes of the doubly linked list together. The usage of the *prev* pointer allows reverse traversal I the doubly linked list.

Structure of Doubly Linked List Data Container

```
template <typename T>
struct LinkedList
{
    DoublyNode<T> *head;
    DoublyNode<T> *tail;
    int size;
```

Figure 4 Structure of the Doubly Linked List Data Container

Above is the implementation of the created doubly linked list data container. The *head* variable stores the pointer to the first element's node of the linked list, whereas the *tail* variable stores the pointer to the last element's node in the linked list. The tail pointer is used to allow reverse traversal, and functions such as insert at end and get at index can be performed more efficiently. The third variable is an integer- *size*. This variable keeps track of

the number of nodes in the doubly linked list so that traversal is not needed to get the size of the linked list.

Review Structure

```
struct Review
{
  string content;
  int userRating;
  DynamicArray<Word> positiveWordsUsed;
  DynamicArray<Word> negativeWordsUsed;
  int positiveWordCount;
  int negativeWordCount;
  double sentimentScore;
```

Figure 5 Structure of Review

To record user input, the code builds a struct called Review with many fields. It has dynamic arrays for both positive and negative words used, numbers for the count of both positive and negative words, and integers for user ratings. Furthermore, it has a double sentiment score, which gives the user's sentiment a numerical value.

Word Structure

```
struct Word
{
  string wordString;
  int usedFrequency;
```

Figure 6 Structure of Word

This code defines a two-member structure called 'words': 'wordString' is a string to store the actual words, while 'usedFrequency' an integer to track the number of times the word has been used.

2.1.2 Algorithms

Linear Search Algorithm

```
// Function to find positive and negative words in review using linear search algorithm
void linearSearchAnalyzeReview(
    const DynamicArray<Word> &negativeWordsList, const DynamicArray<Word> &positiveWordsList)
{
  stringstream ss(this->content);
  string word;

  while (ss >> word)
  {
    bool found = false;

    // lowercase and remove punctuation
    transform(word.begin(), word.end(), word.begin(), ::tolower);
    word.erase(remove_if(word.begin(), word.end(), [](unsigned char c)
                         { return ispunct(c); }),
               word.end());

    // Check if it is duplicate word in review
    bool duplicateWord = false;
    for (int i = 0; i < negativeWordsUsed.getSize(); i++)
    {
      if (negativeWordsUsed.getAtIndex(i).wordString == word)
      {
        duplicateWord = true;
        break;
      }
    }
    if (!duplicateWord)
    {
      for (int i = 0; i < positiveWordsUsed.getSize(); i++)
      {
        if (positiveWordsUsed.getAtIndex(i).wordString == word)
        {
          duplicateWord = true;
          break;
        }
      }
```

Figure 7 Analyze Review function implementing linear search algorithm

```
    // Check if it is -ve or +ve
    if (!duplicateWord)
    {
      for (int i = 0; i < negativeWordsList.getSize(); ++i)
      {
        if (negativeWordsList.getAtIndex(i).wordString == word)
        {
          // is negative
          negativeWordsUsed.append(Word(word));
          negativeWordsList.getAtIndex(i).incrementUse();
          found = true;
          break;
        }
      }
      if (!found)
      {
        for (int i = 0; i < positiveWordsList.getSize(); ++i)
        {
          // is positive
          if (positiveWordsList.getAtIndex(i).wordString == word)
          {
            positiveWordsUsed.append(Word(word));
            positiveWordsList.getAtIndex(i).incrementUse();
            found = true;
            break;
          }
        }
      }
    }
  }
  // update -ve and +ve word counts
  this->negativeWordCount = negativeWordsUsed.getSize();
  this->positiveWordCount = positiveWordsUsed.getSize();
  // Calculate sentiment score
  calculateSentimentScore();
}
```

Figure 8 Analyze Review function implementing linear search algorithm (2)

In figure 7 and 8, it displays the code snippets of the linearSearchAnalyzeReview function. This function analyses reviews to identify positive and negative words using a linear search algorithm. It converts every word to lowercase and removes punctuation before checking for duplicates in the previously identified lists. If a word is not duplicate, it will search through the provided negative and positive word lists. When a match is found, it appends the word to the respective list and increments the usage count. Finally, it updates the counts of negative and positive words and calculates the sentiment score based on these counts.

Binary Search Algorithm

```
// Function to find positive and negative words in review using binary search algorithm
void binarySearchAnalyzeReview(DynamicArray<Word> &negativeWordsList, DynamicArray<Word> &positiveWordsList)
{
  stringstream ss(content);
  string word;
  int positiveCount = 0, negativeCount = 0;

  while (ss >> word)
  {
    // Convert the word to lowercase
    transform(word.begin(), word.end(), word.begin(), ::tolower);
    // Remove punctuation
    word.erase(remove_if(word.begin(), word.end(), [](unsigned char c)
                         { return ispunct(c); }),
               word.end());

    // check if it is duplicate word
    bool isDuplicate = !((Word::binarySearch(this->negativeWordsUsed, word) == -1) && (Word::binarySearch(this->positiveWordsUsed, word) == -1)
    if (!isDuplicate)
    {
      // Binary search for negative words
      int negIndex = Word::binarySearch(negativeWordsList, word);

      if (negIndex != -1)
      {
        // If the word is negative
        insertWordInAlphabeticalOrder(negativeWordsUsed, Word(word));
        negativeWordsList.getAtIndex(negIndex).incrementUse();
        negativeCount++;
      }
```

Figure 9 Analyze Review function implementing binary search algorithm

```
      else if (negIndex == -1)
      {
        // Binary search for positive words
        int posIndex = Word::binarySearch(positiveWordsList, word);
        if (posIndex != -1)
        {
          // If the word is positive
          insertWordInAlphabeticalOrder(positiveWordsUsed, Word(word));
          positiveWordsList.getAtIndex(posIndex).incrementUse();
          positiveCount++;
        }
      }
    }
  }

  this->negativeWordCount = negativeWordsUsed.getSize();
  this->positiveWordCount = positiveWordsUsed.getSize();
  calculateSentimentScore();
}
```

Figure 10 Analyze Review function implementing binary search algorithm (2)

Figures 9 and 10 show the code snippets of the binarySearchAnalyzeReview function. It analyses a review to identify positive and negative words using a binary search algorithm for improved efficiency. The function will first check for any duplicates in previously identified lists using a binary search. If a word is not duplicate, it performs a binary search on the negative words list. If found, the word is added to the negativeWordsUsed list then its count is incremented and update the negative count. If not found, it searches the positive words list

similarly. Finally, the function updates the counts of negative and positive words and calculates the sentiment score based on these counts.

Bubble Sort Algorithm

Bubble sort for Linked List

```
void sort(function<bool(const T &, const T &)> compare)
{
  if (size <= 1)
    return;

  bool swapped;
  DoublyNode<T> *left;
  DoublyNode<T> *right = nullptr;

  do
  {
    swapped = false;
    left = head;

    while (left->next != right)
    {
      if (compare(left->next->data, left->data))
      {
        // Swap node pointers instead of swapping data
        DoublyNode<T> *tempNode = left->next;
        left->next = tempNode->next;
        tempNode->next = left;
```

Figure 11 sort() function in the Doubly Linked List Data Container

```
        // check if it is head node
        if (left->prev != nullptr)
        {
          tempNode->prev = left->prev;
          left->prev->next = tempNode;
        }
        else
        {
          tempNode->prev = nullptr;
          head = tempNode;
        }

        left->prev = tempNode;

        // check if it is tail node
        if (left->next != nullptr)
        {
          left->next->prev = left;
        }
        else
        {
          tail = left;
        }
```

Figure 12 sort() function in the Doubly Linked List Data Container (2)

```
        swapped = true;
      }
      else
      {
        left = left->next;
      }
    }
    right = left;
  } while (swapped);
}
```

Figure 13 sort() function in the Doubly Linked List Data Container (3)

Above shows the code applying bubble sort on the doubly linked list data container created. As it moves through the list, it compares neighboring nodes and, if they are out of

order, switches their locations. The team has tested the performance of exchanging pointers and exchanging data, with the performance of exchanging pointers resulting in significantly faster sorting (execution time of 900ms for exchanging data, 130ms for exchanging pointers when sorting word lists). Therefore, for efficiency, it modifies node pointers rather than exchanging data. Special instances involving head and tail nodes are handled by the algorithm, which updates their connections accordingly. This producer is repeated, going over the list several times until no more swaps are required, signifying that the list is completely sorted.

Bubble sort for dynamic array

```
void sort(function<bool(const T&, const T&)> compare){
  if (size <= 1) return;

  for (int i = 0; i< size - 1; i++){
    for (int j = 0; j < size - i - 1; j++){
      if(compare(dataArray[j+1], dataArray[j])){
        //swap elements
        T temp = dataArray[j];
        dataArray[j] = dataArray[j+1];
        dataArray[j+1] = temp;
      }
    }
  }
}
```

Figure 14 Sort function for dynamic array data container.

The above figure shows the sort() function defined in the dynamic array structure. It receives a comparator function and sorts the elements of the array according to the custom comparator. It utilizes a bubble sort algorithm and will loop through all elements of the array until all elements are sorted.

```
static DynamicArray<Word> sortWordsByFrequency(DynamicArray<Word> WordsArray, bool isAscending)
{
  WordsArray.sort([isAscending](const Word &a, const Word &b)
                  { return isAscending ? a.usedFrequency < b.usedFrequency : a.usedFrequency > b.usedFrequency; });
  return WordsArray;
}

static void sortBothWordArrays(DynamicArray<Word> &negativeWordArray, DynamicArray<Word> &positiveWordArray)
{
  auto start = chrono::high_resolution_clock::now(); // Start time

  // Sort the arrays
  sortWordsByFrequency(negativeWordArray, true);
  sortWordsByFrequency(positiveWordArray, true);

  auto end = chrono::high_resolution_clock::now();                                  // Stop time
  auto duration = chrono::duration_cast<chrono::milliseconds>(end - start).count(); // Measure duration
  // cout << "---Execution time for sorting positive and negative words lists using arrays: " << duration << " ms" << endl;
}
```

Figure 15 Usage of the sort() function to sort words by usedFrequency parameter.

Figure 15 shows one of the functions applying bubble sort in the final program. This function sorts the words in the provided words list according to the *usedFrequency* variable of each Word object in the array. The function shown above used in the final program, is applying the Dynamic Array approach as data containers for Word objects. This function applies the *sort*() function by providing a custom comparator. By using a lambda function, the algorithm can dynamically switch between ascending and descending order based on the *isAscending* flag. The *sortBothWordArrays* function calls *sortWordsByFrequency* for both negative and positive word arrays, sorting them in ascending order while also measuring the execution time with the chrono library.

2.1.3 Final Program Functions

Sentiment Scoring

```
void calculateSentimentScore()
{
  // Calculate sentiment score based on unique positive and negative word counts
  int rawSentimentScore = positiveWordCount - negativeWordCount;
  int N = positiveWordCount + negativeWordCount; // Total number of unique words found
  int minRawScore = -N;
  int maxRawScore = N;
  double normalizedScore = 0.0;

  if (maxRawScore != minRawScore)
  {
    normalizedScore = (double)(rawSentimentScore - minRawScore) / (maxRawScore - minRawScore);
  }

  double sentimentScore = 1 + (4 * normalizedScore);
  this->sentimentScore = round(sentimentScore * 100.0) / 100.0; // Rounded sentiment score
}
```

Figure 16 Source code for the calculateSentimentScore() function

Figure 16 shows the calculateSentimentScore function that computes a sentiment score based on the counts of unique positive and negative words identified in a text. It calculates a raw sentiment score by subtracting the negative word count from the positive word count. The function also determines the total number of unique words found (N) to define the minimum and maximum possible raw scores. A normalized score is then calculated to scale the raw sentiment score between 0 and 1, using the minimum and maximum raw scores as boundaries. Finally, the normalized score is transformed into a sentiment score on a scale from 1 to 5 and rounded to two decimal places.

Analyze Review

```
void static binarySearchAnalyzeAllReviews(
    LinkedList<Review> &allReviews, DynamicArray<Word> &negativeWordList, DynamicArray<Word> &positiveWordList
)
{
  DoublyNode<Review> *current = allReviews.head;
  while (current != nullptr)
  {
    current->data.binarySearchAnalyzeReview(negativeWordList, positiveWordList);
    current = current->next;
  }
}

void static linearSearchAnalyzeAllReviews(
    LinkedList<Review> &allReviews, DynamicArray<Word> &negativeWordList, DynamicArray<Word> &positiveWordList
)
{
  DoublyNode<Review> *current = allReviews.head;
  while (current != nullptr)
  {
    current->data.linearSearchAnalyzeReview(negativeWordList, positiveWordList);
    current = current->next;
  }
}
```

Figure 17 Source code for the binarySearchAnalyzeAllReviews() function

Above code snippet shows the implementation of two variations of analyze review functions. Both functions loop through all reviews and call the analyze function on each Review object. However, one of the functions uses the linear search approach, and the other uses the binary search approach when analyzing all reviews. Both functions results in the same output, where all reviews' positive and negative words used are identifies, followed by the calculation of the sentiment score. The positive words list, and negative word list provided as parameters in the function also gets updated with each word's used frequency reflecting the frequency of uses in all reviews.

Show Review Analysis

```
void showReviewAnalysis(DynamicArray<Word> &negativeWordList, DynamicArray<Word> &positiveWordList)
{
  if (this->positiveWordCount == 0 && this->negativeWordCount == 0)
  {
    binarySearchAnalyzeReview(negativeWordList, positiveWordList);
  }

  // Print out the results
  cout << "\n\n--- SHOWING SPECIFIC REVIEW ANALYSIS\n";
  cout << "\nUser Review:" << endl;
  cout << this->content << endl;
  cout << endl;

  cout << "Positive Words : " << positiveWordCount << endl;
  for (int i = 0; i < positiveWordCount; i++)
  {
    cout << "  -" << positiveWordsUsed.getAtIndex(i).wordString << endl;
  }
  cout << "\nNegative Words : " << negativeWordCount << endl;
  for (int i = 0; i < negativeWordCount; i++)
  {
    cout << "  -" << negativeWordsUsed.getAtIndex(i).wordString << endl;
  }

  cout << "\nUser given rating is: " << to_string(userRating) << endl;
  cout << "Sentiment score is: " << sentimentScore << ", thus the rating should be equal to " << static_cast<int>(round(sentimentScore)) << endl;
  cout << "\nAnalysis Output: " << endl;
  if (userRating == static_cast<int>(round(sentimentScore)))
  {
    cout << "User's subjective evaluation matches the sentiment score provided by the analysis. There is a consistency between the sentiment score generated by the analysis and the
    user's evaluation of the sentiment." << endl;
  }
  else
  {
    double scoreDiff = round(abs(sentimentScore - userRating) * 100.0) / 100.0;
    if (scoreDiff < 1)
    {
      cout << "User's subjective evaluation does not match the sentiment score provided by the analysis. There is a small inconsistency (" << fixed << setprecision(2) << abs(scoreDiff)
      << " difference only) between the sentiment score generated by the analysis and the user's evaluation of the sentiment." << endl;
    }
    else
    {
      cout << "User's subjective evaluation does not match the sentiment score provided by the analysis. There is a great inconsistency (" << fixed << setprecision(2) << abs(scoreDiff)
      << " difference) between the sentiment score generated by the analysis and the user's evaluation of the sentiment." << endl;
    }
  }
}
```

Figure 18 Source code for the showreviewAnalysis() function

The above code snippet shows the function used to display the analyzed results of a particular review. This function first displays the review content. Then, it shows the list of positive and negative words used, together with the total count of positive and negative words. It also compares the difference (if any) between the calculated sentiment score, and user provided rating. Finally, an appropriate analysis of output is displayed according to the differences found.

View Reviews by Rating

```
void viewReviewsByRating(LinkedList<Review> &allReviews)
{
  // User select desired rating 1-5 to filter
  std::cout << "\nPlease select a rating from 1 to 5." << endl;
  int selectedRating = promptIntInput(1, 5);

  // pagination
  int increment = 10; // number of reviews per page
  // int shownCount = 0;
  int index = 1;
  bool nextPage = true;
  bool reachedEnd = false;
  int currIndex = 0;
  DoublyNode<Review> *current = allReviews.head;
  while (nextPage && current != nullptr)
  {
    int shownCount = 0;
    std::cout << endl;
    while (current != nullptr && shownCount < increment)
    {
      index++;
      if (current->data.userRating == selectedRating)
      {
        std::cout << "\nReview #" << to_string(index) << endl;
        std::cout << "Content: " << endl;
        std::cout << current->data.content << endl;
        shownCount++;
      }
      current = current->next;
    }
    if (current == nullptr)
    {
      std::cout << "\n\n---You've reached the end of the list." << endl;
      reachedEnd = true;
    }
    if (!reachedEnd && shownCount == increment)
    {
      std::cout << "\nWould you like to view next page? (1- yes, 2- no)" << endl;
      nextPage = (promptIntInput(1, 2) == 1);
    }
    else
    {
      nextPage = false;
    }
  }
  if (current == nullptr && !reachedEnd)
  {
    std::cout << "\nNo reviews found with the selected rating." << endl;
  }
}
```

Figure 19 Source code for the viewReviewsByrating() function

The viewReviewsByRating function enables users to filter and view reviews based on a selected rating from 1 to 5. It prompts the user for input and sets up pagination variables, including how many reviews to display per page. The function enters a loop where it iterates through the reviews, checking each review's rating against the user's selection. Matching reviews are displayed until the page limit is reached or all reviews have been shown. After

displaying the specified number of reviews, the user is prompted to decide whether to view the next page. The process continues until the end of the list is reached, at which point the user is notified that there are no more reviews to display.

View Sentiment Summary

```
void viewTrends(LinkedList<Review> &allReviews)
{
  int arraySize = allReviews.size;
  if (arraySize == 0)
  {
    std::cout << "No reviews to summarize." << endl;
    return;
  }

  double totalSentimentScore = 0.0;
  double totalUserRating = 0.0;
  int rating1 = 0;
  int rating2 = 0;
  int rating3 = 0;
  int rating4 = 0;
  int rating5 = 0;
  int sentimentScore1 = 0;
  int sentimentScore2 = 0;
  int sentimentScore3 = 0;
  int sentimentScore4 = 0;
  int sentimentScore5 = 0;

  // calculate average and frequency of ratings and sentiment scores
  DoublyNode<Review> *current = allReviews.head;
  while (current != nullptr)
  {
    // const Review &currentReview = allReviews.getAtIndex(i);
    totalSentimentScore += current->data.sentimentScore;
    totalUserRating += current->data.userRating;
    int roundedSentimentScore = static_cast<int>(round(current->data.sentimentScore));
    switch (current->data.userRating)
    {
    case 1:
      rating1++;
      break;
    case 2:
      rating2++;
      break;
    case 3:
      rating3++;
      break;
    case 4:
      rating4++;
      break;
    case 5:
      rating5++;
      break;
    default:
      break;
    };
```

Figure 20 Source code for the viewTrends() function

```
    switch (roundedSentimentScore)
    {
    case 1:
      sentimentScore1++;
      break;
    case 2:
      sentimentScore2++;
      break;
    case 3:
      sentimentScore3++;
      break;
    case 4:
      sentimentScore4++;
      break;
    case 5:
      sentimentScore5++;
      break;
    default:
      break;
    };
    current = current->next;
  }

  double averageSentimentScore = totalSentimentScore / arraySize;
  double averageUserRating = totalUserRating / arraySize;

  // Display results
  std::cout << "\n--- SHOWING RATING AND SENTIMENT SCORE TRENDS" << endl;
  std::cout << "\nTotal reviews = " << to_string(arraySize) << endl;
  std::cout << "Average rating = " << fixed << setprecision(2) << abs(round(averageUserRating * 100.0) / 100.0) << endl;
  std::cout << "Average sentiment score = " << fixed << setprecision(2) << abs(round(averageSentimentScore * 100.0) / 100.0) << endl;
  std::cout << "\nNumber of Reviews with Ratings: " << endl;
  std::cout << "1 star = " << to_string(rating1) << " reviews" << endl;
  std::cout << "2 star = " << to_string(rating2) << " reviews" << endl;
  std::cout << "3 star = " << to_string(rating3) << " reviews" << endl;
  std::cout << "4 star = " << to_string(rating4) << " reviews" << endl;
  std::cout << "5 star = " << to_string(rating5) << " reviews" << endl;
  std::cout << "\nNumber of Reviews with Sentiment Score: " << endl;
  std::cout << "1 star = " << to_string(sentimentScore1) << " reviews" << endl;
  std::cout << "2 star = " << to_string(sentimentScore2) << " reviews" << endl;
  std::cout << "3 star = " << to_string(sentimentScore3) << " reviews" << endl;
  std::cout << "4 star = " << to_string(sentimentScore4) << " reviews" << endl;
  std::cout << "5 star = " << to_string(sentimentScore5) << " reviews" << endl;
}
```

Figure 21 Source code for the viewTrends() function (2)

Figures 20 and 21 are the code snippets for sentiment summary. The viewTrends function summarizes user reviews by analyzing both the user ratings and sentiment scores from an array of Review objects. It calculates the average user rating and sentiment score and counts the number of reviews fall into each rating category (1-5 stars) for both the ratings and sentiment scores. The function iterates through all reviews, accumulating total sentiment scores and user ratings, and then counts how many reviews correspond to each rating level and rounded sentiment score. After processing, it outputs a summary displaying the total number of reviews, average rating, average sentiment score, and the distribution of ratings and sentiment scores across 1 to 5 stars.

View list of positive and negative word frequency

```
case 5: // View list of positive words frequency
{
  cout << "\nWould you like to view in ascending order or descending order? (1- ascending order, 2-descending order)" << endl;
  bool isAscendingOrder = (promptIntInput(1, 2) == 1);
  DynamicArray<Word> sortedPositiveWordArray = Word::sortWordsByFrequency(positiveWordList, isAscendingOrder);
  if (isAscendingOrder)
  {
    cout << "\n--- SHOWING LIST OF POSITIVE WORDS USED IN ASCENDING ORDER:\n\n";
    viewWordsList(sortedPositiveWordArray);
  }
  else
  {
    cout << "\n--- SHOWING LIST OF POSITIVE WORDS USED IN DESCENDING ORDER:\n\n";
    viewWordsList(sortedPositiveWordArray);
  }
  allReviewSubmenu(allReviews, negativeWordList, positiveWordList);
  break;
}
case 6: // View list of negative words frequency
{
  cout << "\nWould you like to view in ascending order or descending order? (1- ascending order, 2-descending order)" << endl;
  bool isAscendingOrder = (promptIntInput(1, 2) == 1);
  DynamicArray<Word> sortedNegativeWordArray = Word::sortWordsByFrequency(negativeWordList, isAscendingOrder);
  if (isAscendingOrder)
  {
    cout << "\n--- SHOWING LIST OF NEGATIVE WORDS USED IN ASCENDING ORDER:\n\n";
    viewWordsList(sortedNegativeWordArray);
  }
  else
  {
    cout << "\n--- SHOWING LIST OF NEGATIVE WORDS USED IN DESCENDING ORDER:\n\n";
    viewWordsList(sortedNegativeWordArray);
  }
  allReviewSubmenu(allReviews, negativeWordList, positiveWordList);
  break;
}
case 7: // Search for a word and view its corresponding sentiment analysis
  searchWord(negativeWordList, positiveWordList);
  allReviewSubmenu(allReviews, negativeWordList, positiveWordList);
  break;
default:
  allReviewSubmenu(allReviews, negativeWordList, positiveWordList);
  break;
}
```

Figure 22 Source code for the submenu selection for view list of positive words and negative words

```
void viewWordsList(DynamicArray<Word> &wordList)
{
  // Pagination
  int increment = 10;
  int start = 0;
  int end = start + increment;
  int max = wordList.getSize();
  bool nextPage = true;
  while (nextPage)
  {
    cout << endl;
    Word::displayTop(wordList, start, end);
    if (start + increment < max)
    {
      start += increment;
      end += increment;
      cout << "\nWould you like to view next page? (1- yes, 2- no)" << endl;
      nextPage = (promptIntInput(1, 2) == 1);
    }
    else
    {
      cout << "You've reached the end of the list." << endl;
      break;
    }
  }
}
```

Figure 23 Source code for the viewWordsList() function

```
  static void displayTop(DynamicArray<Word> &wordsArray, int start, int end)
  {
    // Displays the elements from starting index to end index
    int size = wordsArray.getSize();
    int max = size < end ? size : end;
    for (int i = start; i < max; i++)
    {
      cout.width(20);
      cout << left << wordsArray.getAtIndex(i).wordString << right << " = " << to_string(wordsArray.getAtIndex(i).usedFrequency) << " times" << endl;
    }
  }
};
```

Figure 24 Source code for the displayTop() helper function

Figures 22, 23 and 24 above shows the code snippets of view list of positive and negative word frequency functions. It provides the ability to display the frequency of words used in reviews, specifically focusing on positive and negative sentiment words. The user is first prompted to choose whether to view the list in ascending or descending order. Based on the choice, the program sorts the list of positive or negative words by their frequency using the sortWordsByFrequency function. The sorted list is then displayed in pages of ten words, utilizing the viewWordsList and displayTop functions to show the words along with their usage frequency. Users can navigate through the list page by page, with the option to continue viewing or stop once they have reached the end of the list. This feature allows users to analyze the most or least frequent words in a structured manner.

Search for a word and view its sentiment analysis

```
void searchWord(DynamicArray<Word> &negativeWordList, DynamicArray<Word> &positiveWordList)
{
  // Get string to search and transform into lowercase
  string searchKey;
  cout << "\n--- SEARCH WORD";
  cout << "\nEnter word to search: ";
  getline(cin, searchKey);
  transform(searchKey.begin(), searchKey.end(), searchKey.begin(), ::tolower);

  // Check if is negative word
  int negativeIndex = Word::binarySearch(negativeWordList, searchKey);
  cout << "\nSearching..." << endl;
  if (negativeIndex != -1)
  {
    cout << "\nSearch found!" << endl;
    cout << "\nThe word \"" << searchKey << "\' is a negative sentiment word." << endl;
    cout << "It has been used " << to_string(negativeWordList.getAtIndex(negativeIndex).usedFrequency) << " times in all reviews." << endl;
    // return;
  }
  else
  {
    // Check if is positive word
    int positiveIndex = Word::binarySearch(positiveWordList, searchKey);
    if (positiveIndex != -1)
    {
      cout << "\nSearch found!" << endl;
      cout << "\nThe word \"" << searchKey << "\' is a positive sentiment word." << endl;
      cout << "It has been used " << to_string(positiveWordList.getAtIndex(positiveIndex).usedFrequency) << " times in all reviews." << endl;
      // return;
    }
    else
    {
      // Not -ve or +ve word
      cout << "\nThe word \"" << searchKey << "\' is neither a positive or negative sentiment word." << endl;
    }
  }

  // Check if would like to search for word again
  cout << "\nWould you like to search for another word? (1- yes, 2- no)" << endl;
  bool searchAgain = (promptIntInput(1, 2) == 1);
  if (searchAgain)
  {
    searchWord(negativeWordList, positiveWordList);
  }
  else
  {
    return;
  }
}
```

Figure 25 Source code for the searchWords() function

The searchWord function in figure 25 allows users to search for a word in two separate lists which are one containing negative sentiment words and the other containing positive sentiment words. It will prompt users to enter a word, which will be converted to lowercase for consistent searching. First, the function performs a binary search on the negative word list. If the word is found, it displays the word's usage frequency in reviews as a negative sentiment word. If not found, it will then perform a binary search on the positive word list and reports the word's frequency if it is a positive sentiment word. If the word is not found in either list, the function indicates that the word has no sentiment association. The user is then prompted to search for another word or exit the function.

Prompt user for input

```
int promptIntInput(int start, int end)
{
  bool validInput = false;
  string input;

  while (!validInput)
  {
    cout << "--Enter a choice between " << start << " and " << end << ": ";
    string input;
    getline(cin, input);

    try
    {
      int choice = stoi(input);
      if (choice < start || choice > end)
      {
        cout << "Invalid input. ";
      }
      else
      {
        validInput = true;
        return choice;
      }
    }
    catch (const invalid_argument &e)
    {
      cout << "Invalid input. ";
    }
    catch (const out_of_range &e)
    {
      cout << "Invalid input. ";
    }
  }
  return -1;
}
```

Figure 26 Source code for the promptIntInput() function

Throughout the final developed program, navigation is done by the user using integer inputs as menu selections. Therefore, the above code snippet shows the reusable function used to prompt users for integer inputs complete with validation and range checks. This function receives the start and end index ranges that act as the user input boundaries. The function then checks if the user's input is valid. This function utilizes a while loop to repeat prompting the user to input until a valid input is received.

Pagination

The system utilizes pagination when displaying a long list of records. The following are two implementations of pagination in the final system.

```
// pagination
int increment = 10; // number of reviews per page
int index = 1;
bool nextPage = true;
bool reachedEnd = false;
int currIndex = 0;
DoublyNode<Review> *current = allReviews.head;
while (nextPage && current != nullptr)
{
  int shownCount = 0;
  std::cout << endl;
  while (current != nullptr && shownCount < increment)
  {
    index++;
    if (current->data.userRating == selectedRating)
    {
      std::cout << "\nReview #" << to_string(index) << endl;
      std::cout << "Content: " << endl;
      std::cout << current->data.content << endl;
      shownCount++;
    }
    current = current->next;
  }
  if (current == nullptr)
  {
    std::cout << "\n\n---You've reached the end of the list." << endl;
    reachedEnd = true;
  }
  if (!reachedEnd && shownCount == increment)
  {
    std::cout << "\nWould you like to view next page? (1- yes, 2- no)" << endl;
    nextPage = (promptIntInput(1, 2) == 1);
  }
  else
  {
    nextPage = false;
  }
}
if (current == nullptr && !reachedEnd)
{
  std::cout << "\nNo reviews found with the selected rating." << endl;
}
```

Figure 27 Example implementation of pagination features in the final system (show revies by rating)

The above figure shows the application of pagination in the show reviews by rating feature of the system. According to the user's selected rating to view, it starts at the head of the linked list and checks each review's rating. Utilizing a while loop and appropriate counter, for every ten reviews that match the selected rating, the user is asked whether they want to continue to the next page. When the end of the list is reached, a message is displayed. If no reviews with the selected rating are found, the user is informed.

```
void viewWordsList(DynamicArray<Word> &wordList)
{
  // Pagination
  int increment = 10;
  int start = 0;
  int end = start + increment;
  int max = wordList.getSize();
  bool nextPage = true;
  while (nextPage)
  {
    std::cout << endl;
    Word::displayTop(wordList, start, end);
    if (start + increment < max)
    {
      start += increment;
      end += increment;
      std::cout << "\nWould you like to view next page? (1- yes, 2- no)" << endl;
      nextPage = (promptIntInput(1, 2) == 1);
    }
    else
    {
      std::cout << "You've reached the end of the list." << endl;
      break;
    }
  }
}
```

Figure 28 Example implementation of pagination features in the final system (show words list)

Above code snippet shows another pagination implementation in the final program. It uses integer variables that act as counter and range indexes. It also uses the displayTop() helper function that will print out the elements from the word list according to specified starting and ending range indexes. Using a while loop, each page is displayed with the size of 10-word records per page. After displaying each page, the user is prompted whether they want to view the next page or exit current function.

2.2 System Input Output Screenshots

2.2.1 Main Menu

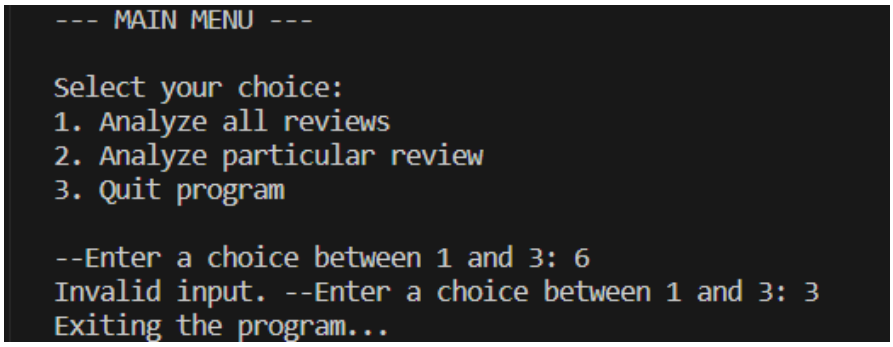


Figure 29 Main Menu of the Program

The figure above shows the main menu display on the startup of the program. It displays three main menu options and a prompt for the user to choose an action. If the user enters "1", the program will analyze all reviews. If the user enters "2", the program will analyze a specific review. If the user enters "3", the program will execute an exit command and the program will end. If a number other than 1, 2 and 3 is entered, a validation message "Invalid input." will be shown, prompting user to choose an action again.

2.2.2 Analyze All Reviews

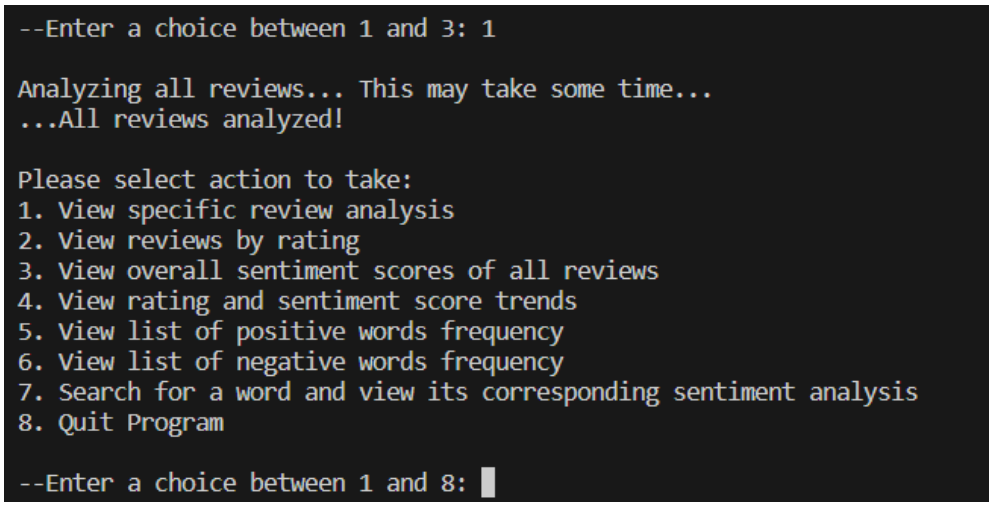


Figure 30 Submenu for "Analyze All Reviews"

The figure above shows the submenu with additional options for further analysis after the option for "Analyze all reviews" is selected. The program performs functions such as iterating through all reviews, positive and negative word lists, analyzing all reviews using binary search on both positive and negative word lists, and calculating sentiment scores for each review. After all reviews have been analyzed, a confirmation message will be shown.

Then, a submenu with various options will be displayed, prompting the user to choose an action. If the user enters an invalid input, a validation message “Invalid input.” will be shown, prompting user to choose an action again. Below is the breakdown of each option:

1. **View specific review analysis:** This lets the user select a specific review to analyze in detail. The user can see sentiment scores, word frequency, and other details for the chosen review.
2. **View reviews by rating:** This option allows the user to filter and view reviews based on specific ratings. For example, they could choose to see only the 5-star reviews or the 1-star reviews, giving insight into reviews categorized by user ratings.
3. **View overall sentiment scores of all reviews:** This provides a summary of sentiment scores across all reviews. The user can see how reviews trend in terms of positive or negative sentiment.
4. **View rating and sentiment score trends:** This displays trends over time or across reviews, showing how ratings and sentiment scores change. It might reveal patterns, like whether sentiment improves or declines over time.
5. **View list of positive words frequency:** This shows a frequency list of positive words used in all reviews, helping the user see which positive words appear most or least often.
6. **View list of negative words frequency:** Like the previous option, this lists negative words by frequency, showing which negative words are most or least common in the reviews.
7. **Search for a word and view its corresponding sentiment analysis:** This allows the user to input a specific word and see its sentiment context and frequency within the reviews. The user can find out how often a word appears and whether it is associated with positive or negative reviews.
8. **Quit Program:** This option ends the program, closing the review analysis.

2.2.3 View Specific Review Analysis

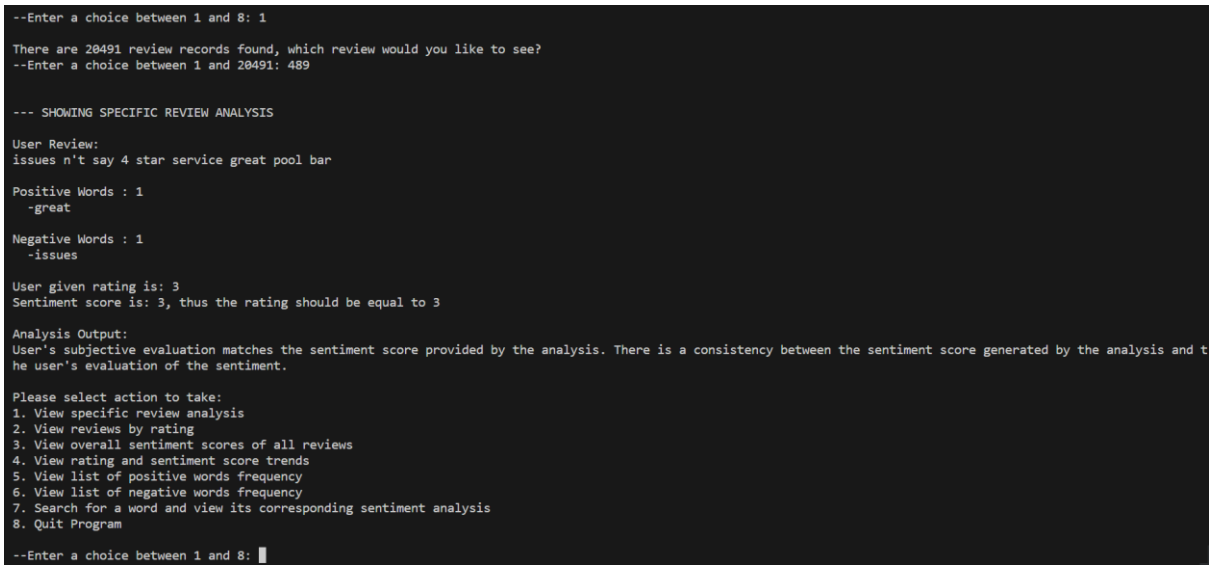


Figure 31 Code Output for View Specific Review Analysis

The figure above shows the code output for "View specific review analysis". The program prompts the user to enter a number within a valid range to select a specific review to analyze. Once the user enters a valid number, the program fetches the relevant review data, such as the review content, positive and negative word lists, user rating, sentiment score, and its analysis output. Then, the submenu for “Analyze all reviews” is shown, prompting the user to choose an action again.

2.2.4 View Reviews by Rating

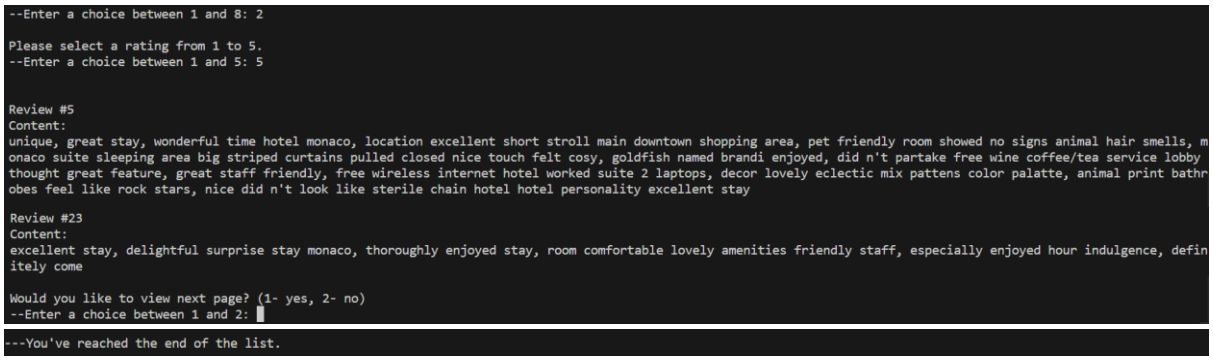


Figure 32 Code Output for View Reviews by Rating

The figure above shows the output code snippets for viewing reviews based on user ratings. The program prompts the user to enter a rating number between 1 to 5. Once a valid rating number is entered, the program will display ten reviews at a time with the rating number the user entered. The program then prompts the user to choose whether to view the next page. If the user enters “1”, the next ten reviews will be shown. If the user enters “2”, the submenu for “Analyze all reviews” is shown, prompting user to choose an action again. If

all reviews have been shown, the program will inform the user that there are no more reviews to display. Then, the submenu for “Analyze all reviews” is shown, prompting user to choose an action again.

2.2.5 View Overall Sentiment Scores of All Reviews

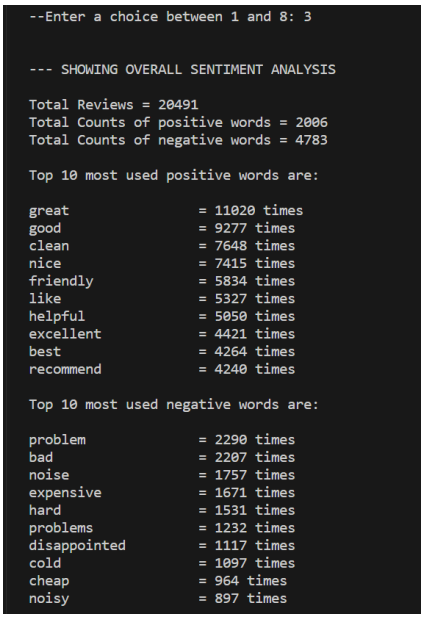


Figure 33 Code Output for Viewing Overall Sentiment Scores of All Reviews

The figure above shows the output code for viewing overall sentiment scores of all reviews. The program displays the overall sentiment analysis that are being calculated by the program. After processing the calculations and sorting, summaries such as the total number of reviews, total word count of positive and negative words found, and the top ten positive and negative words used are shown in this analysis.

2.2.6 View Rating and Sentiment Score Trends

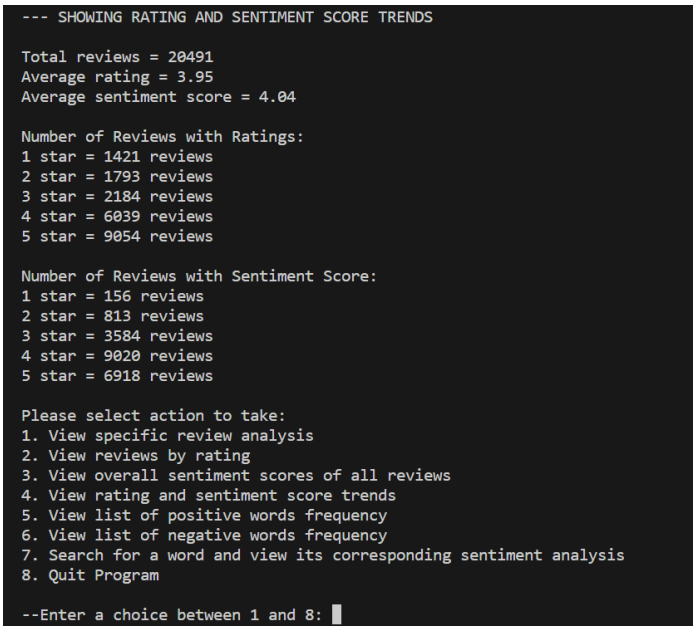


Figure 34 Code Output for Viewing Rating and Sentiment Score Trends

The figure above shows the output code for viewing rating and the sentiment score trends. After some calculations, the program displays the total review, average rating, and average sentiment score. It then shows the number of reviews in their respective rating category (1-5 stars) for both the ratings and sentiment scores. Then, the submenu for "Analyze all reviews" is shown, prompting user to choose an action again.

2.2.7 View List of Positive / Negative Words Frequency

```
--Enter a choice between 1 and 8: 5

Would you like to view in ascending order or descending order? (1- ascending order, 2-descending order)
--Enter a choice between 1 and 2: 2

--- SHOWING LIST OF POSITIVE WORDS USED IN DESCENDING ORDER:


great                = 11020 times
good                 = 9277 times
clean                = 7648 times
nice                 = 7415 times
friendly             = 5834 times
like                 = 5327 times
helpful              = 5050 times
excellent            = 4421 times
best                 = 4264 times
recommend            = 4240 times

Would you like to view next page? (1- yes, 2- no)
--Enter a choice between 1 and 2:
```

```
worth-while          = 0 times
worthiness           = 0 times
wowing               = 0 times
wows                 = 0 times
zeal                 = 0 times
zippy                = 0 times
You've reached the end of the list.
```

Figure 35 Code Output for Viewing List of Positive / Negative Words Frequency

The figure above shows the output code for viewing frequencies of positive and negative words. The program prompts the user to choose whether to display the code in ascending or descending order. Then, the program displays ten lists of words in the selected order at a time. The program then prompts the user to choose whether to view the next page. If the user enters “1”, the next ten lists of words will be shown. If the user enters “2”, the submenu for “Analyze all reviews” is shown, prompting user to choose an action again. If all reviews have been shown, the program informs the user that there are no more reviews to display. Then, the submenu for “Analyze all reviews” is shown, prompting user to choose an action again.

2.2.8 Search for a Word and View Its Corresponding Sentiment Analysis

```
--Enter a choice between 1 and 8: 7

--- SEARCH WORD
Enter word to search: GrEaT

Searching...

Search found!

The word "great' is a positive sentiment word.
It has been used 11020 times in all reviews.

Would you like to search for another word? (1- yes, 2- no)
--Enter a choice between 1 and 2: 1

--- SEARCH WORD
Enter word to search: holiday

Searching...

The word "holiday' is neither a positive or negative sentiment word.

Would you like to search for another word? (1- yes, 2- no)
--Enter a choice between 1 and 2: 2

Please select action to take:
1. View specific review analysis
2. View reviews by rating
3. View overall sentiment scores of all reviews
4. View rating and sentiment score trends
5. View list of positive words frequency
6. View list of negative words frequency
7. Search for a word and view its corresponding sentiment analysis
8. Quit Program

--Enter a choice between 1 and 8:
```

Figure 36 Code Output for Searching for a Word and Viewing Its Corresponding Sentiment Analysis

The figure above shows the output code for searching for a word and viewing its sentiment analysis. The program prompts the user to enter a word to search within the positive and negative word lists. If the word is found, the program will inform the user if the searched word is a positive or negative sentiment word and then display the total used frequency of the word. The program can also determine neutral words in cases if the searched word is neither a positive nor negative sentiment word.

2.2.9 Analyze Particular Review

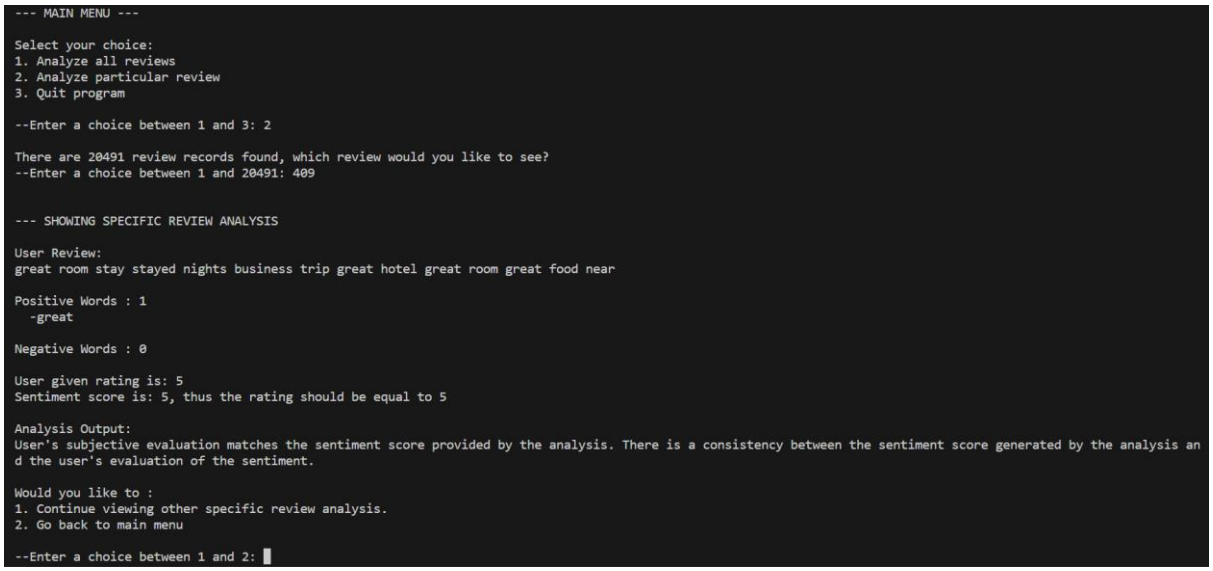


Figure 37 Code Output for Analyze Particular Review

The figure above shows the review analysis of a specific entry after the option for "Analyze particular review" is selected. The input and output are identical to the “View specific analysis review” action. After displaying the selected review's analysis, the program will ask the user if they would like to continue viewing other specific reviews by choosing option “1” or return to the main menu by choosing option “2”. If the user enters an invalid input, a validation message “Invalid input.” will be shown, prompting user to choose an action again.

3. Results and Discussion

3.1 System Efficiency of Containers.

3.1.1 Read Data from Text File

The performance of dynamic arrays and linked lists are measured when reading all review records from the csv and inserting them into the container.

Container	Execution Time (ms)	Space Used (KB)
Dynamic Array (with initial capacity of 1)	166	2359.296
Dynamic Array (with initial capacity equals to number of records)	206	1475.352
Linked List	102	1475.496

Table 1: Comparison of Containers (Storing all review records)

The above table shows the performance of using dynamic array and linked list to read and insert review records from the provided csv file into the respective data containers.

The use of dynamic arrays is tested using two approaches, the first approach involves initializing an array with an initial capacity of 1. Using this approach, the dynamic array undergoes numerous resizing iterations, doubling by capacity each time the size reaches maximum. Whereas another approach initializes a dynamic array with capacity equals the total number of records needed. This is done by looping through the csv file to get total number of lines to initialize the array capacity. Then, the csv file is looped again for reading and appending data into the array container.

By using the first approach (initial capacity of 1), the total space used is significantly bigger than the second approach of using the specific number of records as initial capacity. This is because when the dynamic gets resized, the capacity doubles, thus, resulting in over-allocation of memory which is unused. Whereas the second approach dismisses the need for resizing, using the exact amount of memory allocations needed to store all review records. Apart from that, frequent resizing in the first approach will also cause memory fragmentation and affect system performance (Sazit, 2023). Thus, the second approach is better in terms of space efficiency in terms of total memory occupied, and taking into consideration the potential impact of memory fragmentation on memory utilization.

Time efficiency for both dynamic array approaches is similar. Since arrays involve storing data in contiguous memory, resizing dynamic arrays incurs overhead time for copying data and new memory allocation. Thus, the first approach requires more time to insert new data into the array. However, as the array undergoes multiple resizes, each resizing operation becomes less impactful in time, because it happens less frequently when the capacity doubles. On the other hand, although the second approach eliminates the resizing overhead, this approach requires two loops through the data file- one for counting the number of records and another for inserting the data. The additional time spent in this extra loop causes it to be slower than the first approach.

The use of linked list to read and insert all review records showed the best performance. The use of Linked List showed the same amount of memory occupied with the second approach of dynamic arrays, with a negligible difference of smaller than 1KB. Therefore, it can be concluded that although linked list introduces pointer overheads for the “previous” and “next” pointers for each node, the extra memory used is negligible when compared to dynamic arrays.

The time efficiency for linked list is the best because it involves only one loop through the data file and does not involve the need for resizing. Insertion of a new node takes constant time of O(1), as the memory for each node is allocated separately, there is also no need for shifting elements in linked lists.

In conclusion, for the usage of storing all review records, linked lists are the best approach due to having better speed and space efficiency. The use of dynamic arrays with pre-allocated capacity comes at a close second, and the least efficient approach being the use of dynamic arrays with initial capacity of one.

3.1.2 Linear Search

Container	Execution Time (ms)
Dynamic Array	90249
Linked List	73450

Table 2: Comparison of Dynamic Array and Linked List for Linear Search

Linked List takes 73,450 ms in this operation, while dynamic arrays take 90,249 ms. The Linked List's effective sequential access is the reason for its quicker performance. Both structures, however, exhibit comparatively long execution times, suggesting that linear search takes a long time in general, particularly when dealing with big datasets.

3.1.3 Binary Search

Container	Execution Time (ms)
Dynamic Array	7481
Linked List	106554

Table 3: Comparison of Dynamic Array and Linked List for Binary Search

Dynamic Array performs noticeably better than Linked List, requiring just 7,481 ms as opposed to 106,554 ms for linked List. This significant disparity results from Dynamic Array's rapid random-access capability, which is essential for the strategy of binary search. The lack of direct indexing in Linked List is the reason for the subpar performance.

3.1.4 Bubble Sort

Container	Execution Time (ms)
Dynamic Array	970
Linked List	201

Table 4: Comparison of Dynamic Array and Linked List for Bubble Sort

Remarkably, for bubble sort, Linked List (201 ms) outperforms Dynamic Array (970 ms). This is a result of the numerous switching operations that bubble sort entails, which are more effective in linked lists. This is because sorting in linked lists involves the swapping of pointers only, whereas arrays involve swapping of the whole data. Potential element resizing and shifting during swaps might result in extra overhead for the dynamic array.

3.2 Justification of Data Containers and Algorithms Used

3.2.1 Container for Reviews

The container chosen for reviews is **Linked List**. The main data manipulation applied onto reviews is get at specific index, and insertion. As seen in 2.1.1- Read Data from Text File, Linked List is the most efficient when reading data from the text file and appending the records into the data container. Therefore, Linked List is used to store all review objects in the developed system. Although linked list performs worse when getting the element at specific index compared to dynamic arrays, the use case of getting at specific index is very rare in the developed program. Elements of the review linked list are mostly looped through, which compared to arrays, perform similarly. Thus, Linked List is used for its better time and space efficiency in inserting and storing all review records.

3.2.2 Container for Words

The container used to store word objects is **Dynamic Array**. The main data manipulation functions applied onto word objects is searching. The positive and negative words lists is searched through every word in each review, thus making searching the highest priority for our Word data container efficiently. This is followed by get element at index, sorting, and insertions- in order of highest frequency usage to lowest frequency usage.

The team's findings in 2.1 show that dynamic arrays perform significantly better for binary search algorithm and getting element at index, resulting in a ninety-nine second gap. Whereas dynamic arrays showed weaker performance in the bubble sort algorithm, with a 0.77 second difference. Since the difference in sorting is very small and negligible, dynamic array is chosen as the data container for the Word objects for its better efficiency in handling binary search and get element at index.

3.2.3 Searching algorithm

Combination	Execution Time (ms)
Dynamic Array + Binary Search	7481
Linked List + Linear Search	73450
Dynamic Array + Linear Search	90249
Linked List + Binary Search	106554

Table 5: Comparison of Data Container and its performance with different searching algorithms

The table above concludes the team's findings in 2.1. As seen in the table above, the combination of binary search algorithm with the use of dynamic arrays performs significantly better than the other combinations, with the second fastest combination being 65 seconds slower than the dynamic array + binary search combination. Therefore, the team has decided to apply the **binary search** algorithm for searching from the word lists.

This searching algorithm is used throughout the system in analyzing the sentiment of reviews. For each review, each word is searched for matches in the positive words lists or negative words list. Thus, the efficiency of the searching algorithm is of utmost importance and affects the performance of the developed system.

3.3 Strengths and Weaknesses of the System Developed

3.3.1 Strengths

Speed in analyzing reviews

In the developed program, the analyzing of sentiment words used and the calculation of sentiment scores are done in the same function, thus all reviews are only looped through once to get the complete analysis. Paired with the use of the very efficient binary search algorithm on the dynamic array of Words, the resulting system analyzes all reviews very efficiently and effectively, needing an average of 7.5 seconds to analyze all review records. This result has been very satisfactory and is one of the main strengths of the developed system.

Efficient filtering and pagination

The developed system provides a feature to filter reviews based on rating. This improves the navigation experience of users when faced with a large dataset. Apart from that, the system also implements effective pagination approaches when displaying lists of data. For example, pagination is applied when user views list of positive or negative words used, pagination is also applied when user views list of reviews for a specific rating. Since the dataset involved is big, pagination prevents the user from getting bombarded with outputs and lists and keeps the terminal output tidy. Effective pagination also helps save resources allocation, by only accessing and displaying data when needed. Last but not least, pagination and filtering also provide better user interaction and control when using the developed system.

3.3.2 Weaknesses

Search words according to rating

In the developed final program, words in each review are first searched for a match in the negative words list followed by the positive words list. However, an enhancement can be made to the system, where the program first checks the given user's rating, and categorize it as either positive (4 or 5 stars), neutral (3 stars), or negative (1 or 2 stars). Then, according to the category, words from positive user reviews are searched for positive words list match first, before searching from the negative words list, and vice versa. This is because, when the user gives a positive rating, the tendency to use positive words is higher than the negative word, thus this enhancement can effectively save time in the process of analyzing reviews.

Lack of real-time update of caching for review data

The developed system reads the review data from the CSV file once and loads all reviews into memory. In the case of the dataset growing, and new reviews getting added, the whole system will need to be restarted to get the newest changes in the review data CSV file. This is one of the limitations of the system, as previously analyzed data will need to undergo analyzation again despite being analyzed already. In the case where the system requires real-time updates and integration with live data sources, the system can be enhanced to store analyzed reviews into a new data file and update the result file accordingly when new reviews are added.

4. Conclusion

4.1 Summary of Work

The program developed aims to perform sentiment analysis on hotel reviews using two distinct data structures: dynamic arrays and doubly linked lists. The goal is to compare both approaches and provide insights into the strengths and weaknesses of each in terms of performance and efficiency. Both the array and link list implementation are tested with the use of linear and binary search techniques to identify sentiment words from word lists. The efficiency of the data containers is also tested with the implementation of bubble sort functions for sorting words by frequency. In conclusion, both data containers excel in different scenarios, and the benefits of using both data containers differ according to the specific requirement and implementations of the system. After careful evaluation, future enhancements can be done on the developed system to consider the user rating when performing word sentiment analysis to improve the time efficiency of the system. The system can provide support for the integration of live data sources and allow new reviews to be added within the system.

4.2 Personal Thoughts

The research and development process of this system has borne many fruitful outcomes and knowledge for the team members. By getting hands-on experience and testing with the performance of varying data containers in the system implementations, the team members had the opportunity to learn more in depth about carious data containers' structures, benefits, and setbacks. Analyzing each data containers' performance from both time and space efficiency perspective has also provided a lot of insights into their differences.

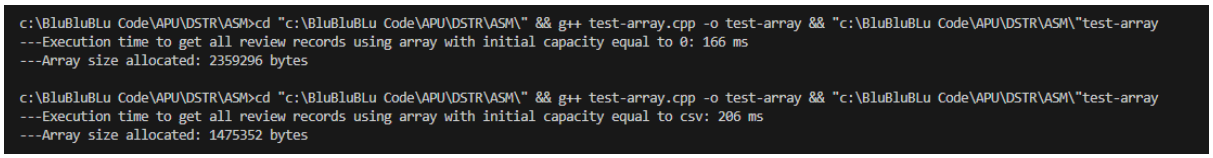
Collaboration within the team also played a key role, as team members contributed unique ideas and worked together to solve complex issues like debugging, thinking of more extra features, and optimizing algorithms. Through this experience, members' understanding of data structures has deepened and the challenges were approached systematically. Additionally, the project highlighted the practical importance of applying the right data structure to real-world applications, and it was a great exposure to all the team members.

5. References

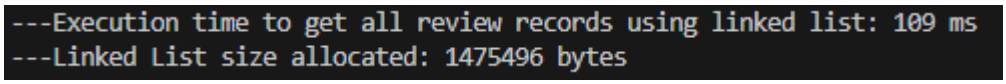
Sazit, E. I. (2023, October 6). *Dynamic Arrays - Ehosanul Islam Sazit - Medium*. Medium; Medium. https://medium.com/@sazitislam96/the-power-dynamic-arrays-comprehensive-guide-3945e1feb907

6. Appendix

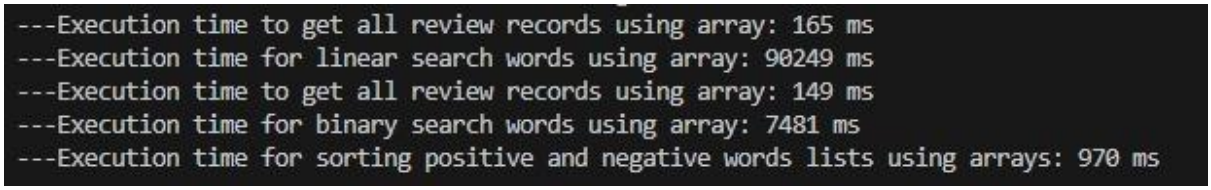
Dynamic Array- time and space efficiency for storing all reviews



Linked List - time and space efficiency for storing all reviews.



Dynamic Array- linear search, binary search, and bubble sort execution time



Linked List- linear search, binary search, and bubble sort execution time