Assignment 3: Data Structures and Algorithms

Construction of a Virtual Rube Goldberg Machine Encompassing All Data Structures

By Priyanka (014), Sarthak (041), Arpita (046), Shravya (055)

ABSTRACT

In this project, a Rube Goldberg machine was designed using various Abstract Data Types (ADTs). The ADTs are coded in C++. Certain set of instructions were listed in the assignment question that required to read a text file and formulate the data collected in different data structures like queue, binary tree, linked list and stack. An API was created and a custom text file with name, date of birth and age of several individuals was read successfully from the command prompt of operating systems like Windows XP or higher.

A virtual Rube Goldberg Machine can successfully incorporate all data structures to do the simple function of reading a file.

TABLE OF CONTENTS

RUBE GOLDBERG MACHINE

What is a Rube Goldberg Machine?

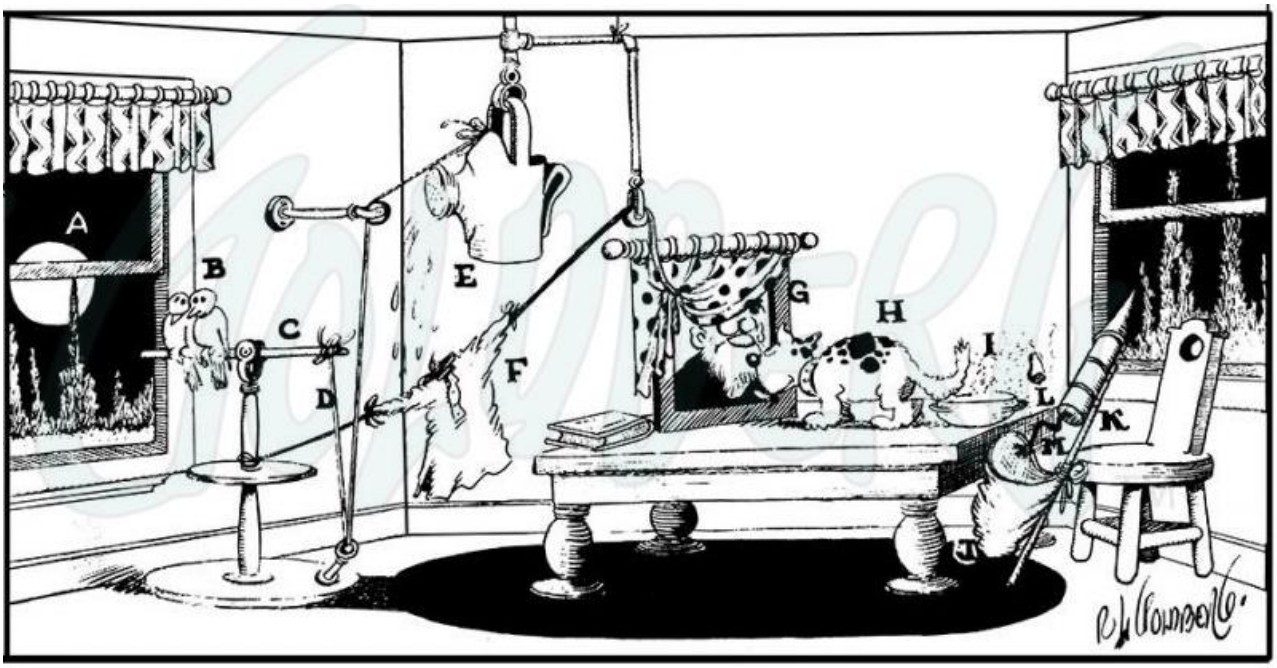
A Rube Goldberg Machine is a complex device that performs simple tasks in indirect and convoluted ways. Reuben Goldberg was an American engineer who changed his career to cartoonist. He was famous for his political cartoons and satire. However, the works which would lead to his lasting fame were called “The Inventions of Professor Lucifer G. Butts”. They involved a character named Professor Lucifer Gorgonzola Butt. In this series, Goldberg drew absurd inventions that would later bear his name: Rube Goldberg Machines. Rube Goldberg received the Pulitzer Prize for his satire in 1948.

Some of Rube Goldberg’s Insane Inventions

The first complex machine that would end up being Goldberg’s most famous invention was his “Automatic Weight Reducing Machine,” drawn in 1914. It used a donut, bomb, balloon and a hot stove to trap an obese person in a room without food, who had to lose weight to get free.

To understand and study more about Rube Goldberg, his granddaughter Jennifer George wrote an in-depth book called “The Art of Rube Goldberg: (A) Inventive (B) Cartoon (C) Genius”. An interesting invention that Jennifer holds dear is called “A Simple Idea for an Automatic Device for Emptying Ash Trays”.

A Simple Idea for an Automatic Device for Emptying Ash Trays



Goldberg’s Ash Tray Emptying Machine

The process of this machine is as follows:

A. A bright full moon that causes
B. Love birds to be romantic and excited. When they get together, their weight
C. Tips the perch which
D. Pulls the string. The can tied to the tree is tipped.
E. The water from the can then falls on the woolen T-shirt
F. And causes it to shrink,
G. Which pulls the curtain tied to the shirt by the string
H. It reveals a portrait that the log is excited by. Hence, the dog wags his tail, which
I. Shakes the tray from its original position.
J. The tray falls into the bag which is filled with asbestos. The asbestos
K. Lights the rocket, which shoots out of the window that takes the tray with it.

Therefore, in eleven highly complicated steps, an ashtray is skyrocketed for the ash to be disposed of. Hence, from the breakdown of this one invention, we can see the pattern for all other inventions of Goldberg. From this analysis, we can see that Rube Goldberg's invention involve **the following elements:**

1. A simple task to be performed
2. Completely unrelated individual mini-processes
3. A lot of circumstantial backing

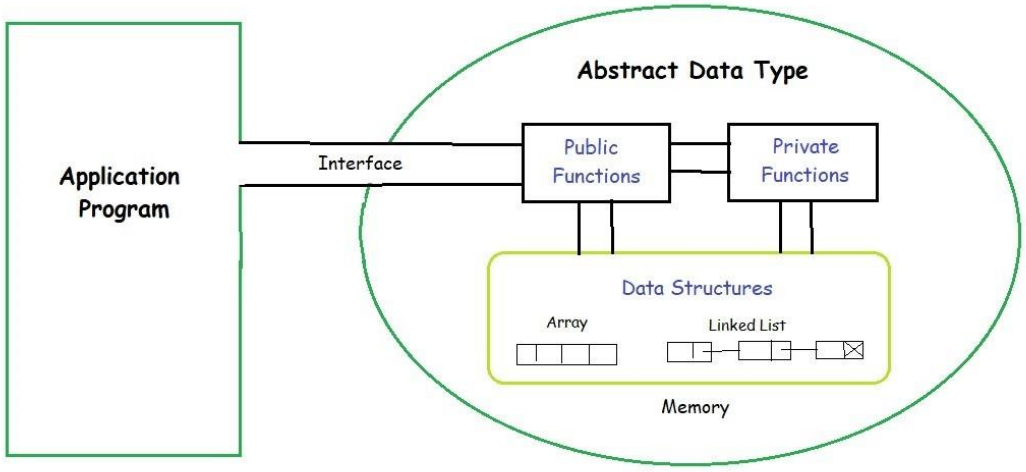
Rube Goldberg Machines were a reflection of Goldberg's satirical cartoons, hence the process of each of his invention is completely bizarre and outlandish. They are made this way to insert irony and humour into a simple comic. It is important to note that all of Rube Goldberg's inventions are physical. They exist to perform physical tasks. In this project, we are required to create a virtual Rube Goldberg Machine.

Our virtual machine takes two key points from the original model: a simple task to be performed, and completely unrelated individual mini-processes. Firstly, there's the task: reading a text file and accepting basic input from the user using ADTs and data structures.

ADTs AND DATA STRUCTURES

What are ADTs and Data Structures?

ADT is abbreviation for Abstract Data Types (or class) for objects whose behavior is user-defined according to the operations to be performed and functions required. An ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view.



How an abstract data type executes itself in a program

ADTs also help in executing one of the four pillars of object-oriented programming called abstraction. Abstraction is the process of hiding unimportant or unnecessary data from the user to reduce chaos while they are interacting with a certain API. Since ADTs do not mention what operations are clearly performed, they do hide some functions and data members as they are being implemented. Hence, they aid in abstraction.

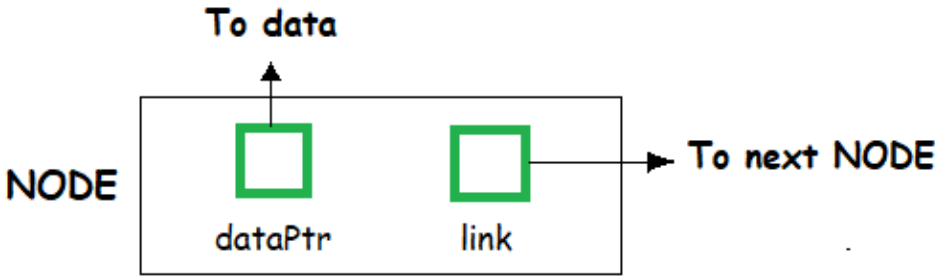
They are the exact opposite of primitive data types like int, float, double etc. where we know the data type of the variable in question. We know what these data types do, and hence how to implement them.

Data structures are concrete ADTs that have proper structures. They can be solidified by code. Some of these data structures are arrays and lists.

Examples of Data Structures

1. List ADT

Commonly known as linked list, this ADT has no particular primitive data type. A list is a chain of blocks of data that are linked by virtue of their addresses. A block of data here is called node. Every node has any amount of data with any primitive data type, and one variable to store the address of the next block, or the next node.

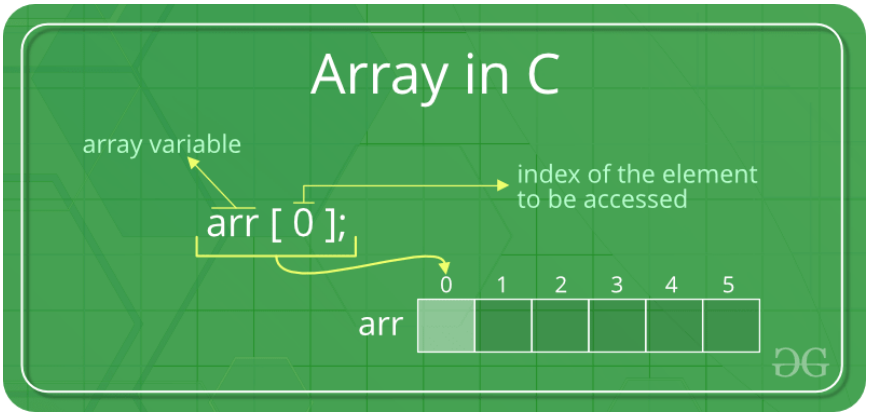


Structure of a block of data/node

Addresses are the way the nodes are linked with each other. Here, the first node is important as it acts as an anchor for the rest of the list.

2. Arrays

Arrays are a predefined collection of data that belongs to the same primitive data type. They exist in all programming languages, and are the first data type to be used whenever multiple (but predefined) number of items belonging to the same data type are required to be used in a program.

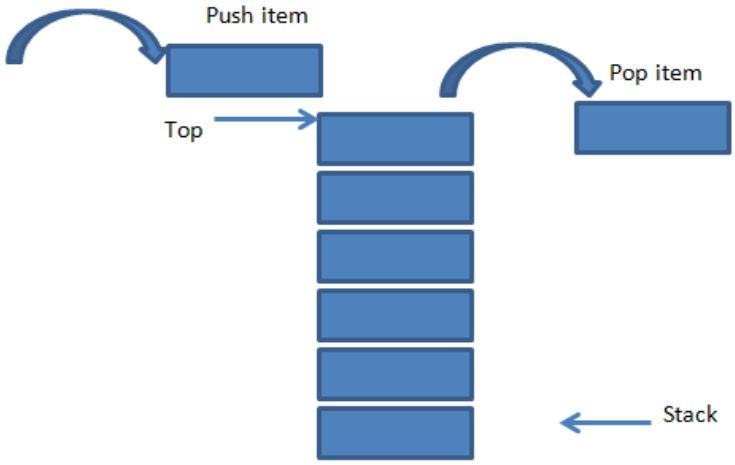


Array indexing

Examples of ADTs

1. Stack ADT

The stack ADT operates on a collection comprised of elements of any proper type T and, like most ADTs, inserts, removes, and manipulates data items from the collection. The stack models the behavior called "LIFO", which stands for "last-in, first-out". The element most recently pushed onto the stack is the one that is at the top of the stack and is the element that is removed by the next pop operation.



Structure of a stack

Stacks, in particular, serve as a basic framework to manage general computational systems as well as in supports of specific algorithms. All modern computers have both hardware and software stacks, the former to manage calculations and the latter to manage runtime of languages such as C++.

Stacks can be coded using another data structure called array. It can also be implemented using a linked list.

```c
#include <stdio.h>

#define MAX 50
int top = -1;
int arr[MAX];

void push(int item) {

    if(top == MAX -1) {
        printf("Stack overflow");
        return;
    } else {
        arr[++top] = item;
        printf("Push element is %d\n", item);
    }
}
```

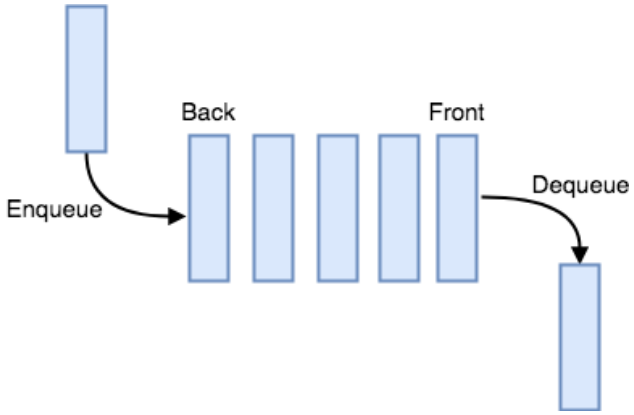
Stack implemented using array

```
/* Write a c program to implement stack using linked list */
#include<stdio.h>   #include<conio.h>      #include<malloc.h>    #include<stdlib.h>
int push();         int pop();      int display();          int choice,i,item;
struct node {
      int data;
      struct node *link;
}*top,*new,*ptr;
main() {            top=NULL;
      printf("\n***Select Menu***\n");
      while(1)  {
            printf("\n1.Push \n2.Pop \n3.Display \n4.Exit\n5.Count");
            printf("\n\nEnter ur choice: ");
            scanf("%d",&choice);
            switch(choice)      {
                  case 1:   push();         break;
                  case 2:   pop();          break;
                  case 3:   display();      break;
                  case 4:   exit(0);
                  case 5:   count();        break;
                  default: printf("\nWrong choice");
            }/* end of switch */
      }/* end of while */
}/* end of main */
```

Stack implemented using linked list

2. Queue ADT

A queue works on FIFO – first in, first out principle. Like all other ADTs, it does not adhere to other primitive data types. It is an ordered list of elements. Hence, its implementation can vary.



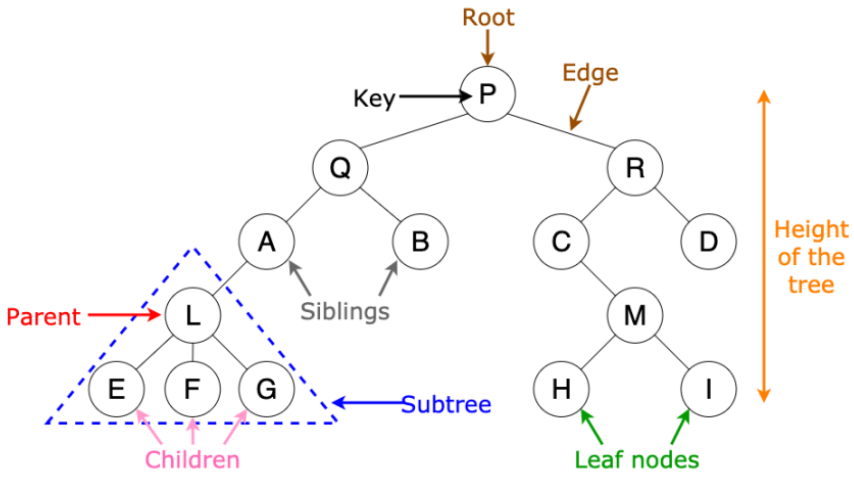
Working of a queue

All these ADTs have certain functions like insert(), remove(), enqueue(), dequeue(), push(), pop(), delete(), display() in common. The changeable nature of these functions along with a varying implementation are what make these structures an Abstract Data Type.

3. Tree ADT

A tree is a widely used abstract data type that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes. Without any kind of restrictions attached to the basic functions of a tree, it can be defined as an ADT. Once this data type is implemented using arrays or linked lists, it becomes a data structure.

In this Rube Goldberg Machine, we use binary tree to store the data for a certain period of time.



Structure of a binary tree

Almost always, a binary tree is implemented using a linked list with a node that contains a variable to preserve the address of related nodes.

DESIGN

Semantically, there are various aspects to the program design that helps to implement abstraction. Mainly, the program is divided into header files, global variables, ADTs, data structures, general display functions, cleaning functions and 11 important functions that help run the entire program.

Header Files

```
1  // * Header files *
2  #include <iostream>
3  #include <fstream>
4  #include <string.h>
5  #include <conio.h>
6  using namespace std;
7  // **
8  // ----------------------------
9  // * Global Variables *
```

All the header files used

Mainly, these header files help with reading the file, taking inputs and outputs as required and converting data types as required.

Global Variables

```
// * Global Variables *

int front=-1, rear=-1, top=-1, ssize=0, sizeofData;
const int size=1200;
char c;

// **
// ----------------------------
```

All the global variables used

These global variables are used in several functions across the API. Most variables are pertaining to the creation of different data structures used in this program.

ADTs

```
// * ADTs *

struct info //for queue (q) and stack (s)
{
    string firstName, lastName, age, dob;
}q[size], s[size];

struct nodeTree //for unordered tree
{
    struct info data;
    struct nodeTree* left;
    struct nodeTree* right;
};

struct nodeLink //for linked list
{
    struct info dataLink;
    nodeLink* next;
}*start, *ptr, *newptr, *r, *prevForInsertion, *ptrForInsertion;

// **
// ------------------------------
```

All the ADTs used

There are three distinct ADTs using the ‘struct’ keyword. Later on, they are implemented as queues, stacks, unordered binary trees and linked lists in the program.

Data Structures

```
// * Framework of Data structures *

void enqueue(struct info element) //queue
{
    if(rear==-1)
    {
        front=rear=0;
        q[rear]=element;
    }
    else
    {
        rear++;
        q[rear]=element;
    }
}

void push(struct info element) //stack
{
    top++;
    s[top]=element;
    ssize++;
}
```

Code for data structures queue and stack used in the program

This program uses four main data structures (queues, stacks, linked lists and binary trees) to implement file handling (namely file reading). There is a certain framework that is declared at

the start of the program. Later on, when data needs to be added to these data structures, the ADTs are modeled in the same manner as the example given in the picture.

General Display Functions

```
// * Display functions *

void displayQueue() //for queue
{
    for(int i=front;i<rear;i++)
    {
        cout<<"First Name: "<<q[i].firstName;
        spaces(q[i].firstName);
        cout<<"\tLast Name: "<<q[i].lastName;
        spaces(q[i].lastName);
        cout<<"\tAge: "<<q[i].age;
        spaces(q[i].age);
        cout<<"\tDate of birth: "<<q[i].dob<<endl;
    }
}

void displayTree(struct info data) //for unordered binary tree
{
    cout<<"First Name: "<<data.firstName;
    spaces(data.firstName);
    cout<<"\tLast Name: "<<data.lastName;
    spaces(data.lastName);
    cout<<"\tAge: "<<data.age;
    spaces(data.age);
    cout<<"\tDate of birth: "<<data.dob<<endl;
    spaces(data.dob);
}
```

Display functions used for queue and binary tree

There are certain generic display functions used in the program to display the data structures after some modification has been done to them. To avoid them from interrupting the important syntactical methods in the program, they have been declared outside the main body of the program.

Cleaning Functions

```
// * Cleaning functions *

void spaces(string a)
//To remove unnecessary spaces and insert necessary spaces
{
    int n=a.length();
    for(int i=0;i<10-n;i++)
    cout<<" ";
}

string removeComma(string str)
//To remove unformatted commas from interfering in the string
{
    int len=str.length();
    if(str[len-1] == ',')
    {
        str[len-1] = '\0';
    }
    return(str);
}

void lineBreak() //To distinguish between outputs
{
    std::cout.fill ('*');
    std::cout.width (127);
    std::cout << " " << '\n';
}
```

All the cleaning functions used

In this program, because of the nature of the data to be input, there are a few functions responsible for extracting the important data from all the unnecessary commas and spaces. These functions are different from the general structure of the program. Hence, they are declared at the start under the title ‘cleaning functions’.

Operative Functions

The entire program is divided into 9 running parts.

```
// * PART 1: Reading input from file *

/* Here, the file name is sample.txt and
there are certain names, ages and dates of birth
that need to be read by our program. */
```

Part 1: Insert from the file into the queue

As the data is read from a file, initially store the data in a queue.

```
// * PART 2: Insert read file elements into a queue data structure *

/* Part 2 consists of two main parts:
dequeuing elements into a stack, display them
and requeing them. */
```

Part 2: Dequeue and requeue

Dequeue each element from the queue, print each item, and requeue each item.

```
// --------------------------------
// * PART 3: Reversing elements using stack *

/* Here, we use pop functions given by stack
to invert the queue and display it. */
```

Part 3: Reverse queue using stack

Reverse the order of the data in the queue by dequeuing each element and pushing them onto a stack. Once all the data is dequeued from the queue and pushed on to the stack, pop off each element of the stack and re-queue each element back into a queue ADT, reversing their order in the queue. Once completed, dequeue each element from the queue and print each item for the user. (Requeue each item to preserve the queue.)

```
// * PART 4: Transferring elements from queue to an unordered binary tree *

/* Here, we insert all the file items from
queue to binary tree. */
```

Part 4: Insert from the queue into a Binary tree

Dequeue the elements from the queue and place them into an unordered binary tree. Add the items into the tree ADT in the order they are dequeued, adhering to the binary tree shape property.

```
// * PART 5: Unordered binary tree traversals *

/* This part deals with exploration related to
preorder and postorder traversals of an unordered binary tree. */
```

Part 5: Pre-order post order of the binary tree

Print the contents of the tree in preorder and print the contents of the tree in post-order.

```
// * PART 6: Insert from a binary tree into Linked list *

/* Here, we shift the elements from an unordered binary tree to
a linked list using inorder traversal. */
```

Part 6: Insert from a binary tree into Linked list-In order traversal

Move the data from the unordered binary tree into a linked list ADT using an in-order traversal of the tree. Print the contents of the list.

```
// * PART 7: Use Quicksort method to sort items in linked list *

/* Here, we employ the quicksort algorithm that depends on pivoting one element
in an array or linked list to sort all the data given in the
current linked list on the basis of first name. */
```

Part 7: Sort and display linked list

Sort the contents of the list using a quick sort in ascending order. Print the contents of the list.

```
// * PART 8: User interaction and input *

/* In this part we take similar input as given in the file
from the user and integrate it into our previously sorted
linked list. */
```

Part 8: Insert into list

Allow the user to interactively enter another name, age, and birthday. Add this to the list in the proper location to maintain the sorted order. Print the contents of the list and ask the user to press any key to continue the processing.

```
// * PART 9: Appending all the functions *

/* This part takes over the entire execution by arranging all the functions
in 8 separate functions. This helps to hide the unimportant information
from the user. */
```

Part 9: Combination

Arrange all the functions in respective combinative functions are check for proper compilation. Integrate the code into our required IDE which is the terminal window.

FUNCTIONS AND OPERATIONS

There are many functions and operations used in this program under the 4 data structures that help run the entire program.

Queue

1. Enqueue() - Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first. Hence, the enqueue() function helps to add data in the start of the queue.

```
void enqueue(struct info element) //queue
{
    if(rear==-1)
    {
        front=rear=0;
        q[rear]=element;
    }
    else
    {
        rear++;
        q[rear]=element;
    }
}
```

Code for adding elements to the queue

2. Dequeue() - When dequeue() is called, the immediate item on the queue is removed from the queue, and then displayed. It is a generic function used for removing elements from the queue.

```
void deQueue1() //Function 2: Used to dequeue, print and requeue each item
{
    struct info temp;
    temp=q[front];
    if(front==rear)
    {
        front=rear=-1;
    }
    else
    {
        front++;
    }
    cout<<"First Name: "<<temp.firstName;
    spaces(temp.firstName);
    cout<<"\tLast Name: "<<temp.lastName;
    spaces(temp.lastName);
    cout<<"\tAge: "<<temp.age;
    spaces(temp.age);
    cout<<"\tDate of birth: "<<temp.dob;

    cout<<endl;
    enqueue(temp);
}
```

Code for dequeing elements from queue

Stack

1. Push() - The push() method allows you to add one or more elements to the end of the array. The push() method returns the value of the length property that specifies the number of elements in the array. If you consider an array as a stack, the push() method adds one or more element at the top of the stack.

```
void push(struct info element) //stack
{
    top++;
    s[top]=element;
    ssize++;
}
```

Code for pushing elements into a stack

2. Pop() - pop() function is used to remove an element from the top of the stack(newest element in the stack). The element is removed to the stack container and the size of the stack is decreased by 1.

```
void pop() //Function 3: To pop elements that will be displayed in reverse order
{
    struct info temp;
    temp=s[top];
    top--;
    enqueue(temp);
}
```

Code for popping elements from stack

Unordered Binary Tree

1. insertTree() – In the program, this function helps to insert elements into the tree by maintaining the binary tree property.

```
void insertTree(nodeTree* x,nodeTree *y) //Function 5: To insert an element into the tree
{
    if(x->left==NULL)
    {
        x->left=y;return;
    }
    else if(x->right==NULL)
    {
        x->right=y;return;
    }
    else
    {
        if(height(x->left)<height(x->right))x=x->left;
        else x=x->right;
        insertTree(x,y);
    }
}
```

Code for inserting node in binary tree

2. Preorder() – This function in the program helps to traverse the tree in preorder method.

```
void preorder(nodeTree* root) //Function 6: For preorder traversal of an unordered binary tree
{
    if(root==NULL)return;
    displayTree(root->data);
    preorder(root->left);
    preorder(root->right);
}
```

Code for preorder traversal

3. Postorder() – This function in the program helps to traverse the tree in postorder method.

```
void postorder(nodeTree* root) //Function 7: For postorder traversal of an unordered binary tree
{
    if(root==NULL)return;
    postorder(root->left);
    postorder(root->right);
    displayTree(root->data);
}
```

Code for postorder traversal

Linked List

1. Insertlink() – This function helps in creating the linked list.

```
void insertlink(nodeLink* np) //Function 9: To insert item into linked list
{
    if(start==NULL)
    {
        start=r=np;
    }
    else
    {
        r->next=np;
        r=np;
    }
}
```

Code to insert new node in linked list

2. quicksort() – This function acts as a driver function to execute the other.

```
void quickSort(nodeLink **headRef) //Function 10: Driver function for quickSort
{
    *headRef=quickSortRecur(*headRef, getTail(*headRef));
    return;
}
```

Driver code for the quicksort recursive function

3. inputFromTerminal() – This function helps to take user input from the terminal window in the same format as the data in the file.

```
void inputFromTerminal() //Function 11: Take input from user in terminal window
{
    string a, b, c, d;
    struct info newdata;
    cout<<"Enter data in the format:"<<endl;
    cout<<"First name, last name, age, date of birth"<<endl;
    cin>>a>>b>>c>>d;
    a=removeComma(a);
    newdata.firstName=a;
    b=removeComma(b);
    newdata.lastName=b;
    c=removeComma(c);
    newdata.age=c;
    d=removeComma(d);
    newdata.dob=d;
    newptr=createlink(newdata);
    placeInLinkedList(newptr);
}
```

Code to take input from terminal window

4. placeInLinkedList() – This function helps to place the input taken by the user into its appropriate position in the list. It traverses the entire list and finds the most suitable place in ascending order.

```
void placeInLinkedList(struct nodeLink* newEntry) //Function 12: Add in sorted linked list
{
    prevForInsertion=start;
    ptrForInsertion=start;
    while(ptrForInsertion!=NULL)
    {
        if(compare(newEntry->dataLink.firstName,ptrForInsertion->dataLink.firstName))
        {
            prevForInsertion=ptrForInsertion;
            ptrForInsertion=ptrForInsertion->next;
        }
        else
        {
            break;
        }
    }
    newEntry->next=ptrForInsertion;
    prevForInsertion->next=newEntry;
}
```

Code to traverse linked list to place element at the right position

Driver Functions

The eight parts are combined into a working program by employing eight functions of the semantic notation void part1(), void part2… void part8().

```
void part1()
{
    lineBreak();
    cout<<"Reading given file."<<endl;
    readFile();
    cout<<endl;
    cout<<"Press any key for processing."<<endl;
    getchar();
}

void part2()
{
    cout<<endl;
    lineBreak();
    cout<<"To dequeue, print and requeue items from file:"<<endl;
    int x=front, y=rear;
    while(x<=y)
    {
        deQueue1(); //Function 2
        x++;
    }
    cout<<"Press any key for processing."<<endl;
    getchar();
}
```

First two functions involved in assimilation of the eight parts

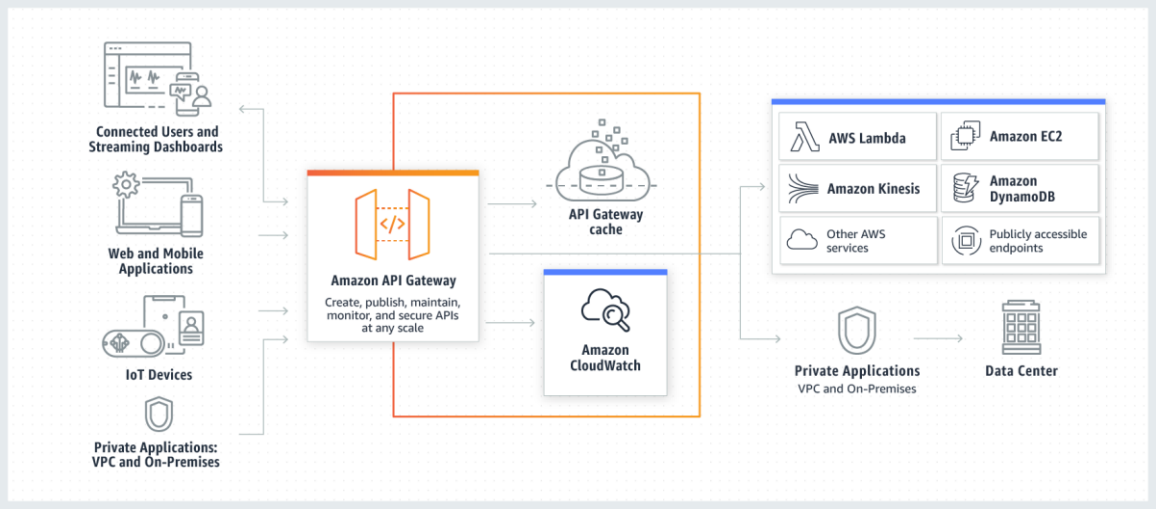
HOW TO USE THE PROGRAM

This complex (and virtual) Rube Goldberg Machine that acts as an API to execute file handling from the operating system.

What is an API?

API is the acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other. Each time an app like Facebook is used, an API is being implemented.

To clarify APIs, an example is important. A person is sitting at a table in a restaurant with a menu of choices to order from. The kitchen is the part of the "system" that will prepare the order. What is missing is the critical link to communicate the order to the kitchen and deliver the food back to the table. That's where the waiter or API comes in. The waiter is the messenger – or API – that takes the request or order and tells the kitchen – the system – what to do. Then the waiter delivers the response back to the table. In this case, the response is the food.



An example of the API for Amazon Payment Getaway

This program, since it does read a text file, can be used as an API in any situation where a text file is needed to be read. However, that is not recommended since the time complexity of this program is rather high. However, this program can be used as a learning opportunity to know as much as one can about data structures and its implementation.

Steps to Implement the Program

1. Download the text file attached with a list of data to be read
2. Open the executable file attached with the report
3. Run the file from your command window (for Windows XP or higher)
4. Read the messages that indicate the processes being conducted on the backend
5. Enter data in the same format as given in the text file when prompted
6. Press given key to exit the API

TIME COMPLEXITY

What is time complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

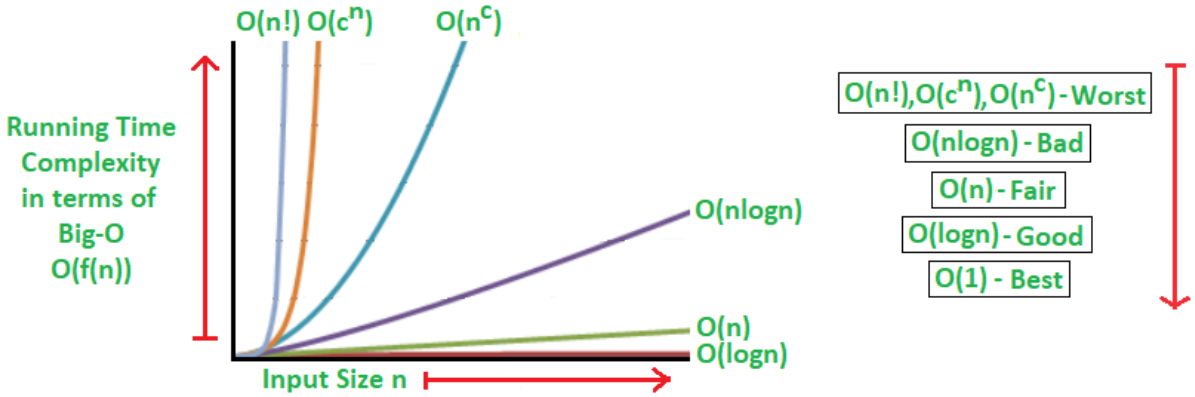
Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n at least and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size.

Time Complexity of algorithm/code is not equal to the actual time required to execute a particular code but the number of times a statement executes. It is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.

Big-O Notation

Big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. It is very easy to compare efficiency of a program using this notation.



Analysis of algorithms

We can find the time complexity of our virtual Rube Goldberg Machine using this notation.

Time Complexity for Rube Goldberg Machine

Function	Code	Complexity
void spaces()	Contains one for loop, called once.	O(n)
void displayQueue()	Contains one for loop, called twice.	O(2n)
void displayLink()	Contains one while loop, called thrice.	O(3n)
void readFile()	Contains one while loop and one for loop.	O(2n)
void enqueue()	Input for queue.	O(1)
void dequeue()	Output for queue.	O(1)
void push()	Input for stack.	O(1)
void pop()	Output for stack.	O(1)
void quicksort()	Quicksort algorithm.	O(n^2)
int compare()	Contains one for loop, called twice.	O(2n)
void placeInLinkedList()	Contains one while loop, called once.	O(n)
void part2()	Contains one while loop, called once.	O(n)
Void part3()	Contains one while loop, called once.	O(n)
void part45and6()	Contains one for loop, called once.	O(n)
void part8()	Contains one while loop, called once.	O(n)

Mathematically, to find the time complexity, we have to add all these separate complexities. Adding them, we get:

Complexity = O(3+15n+n^2)

However, as the value of n increases, the worst time complexity of n^2 starts to overshadow the other value 3+15n. n^2 sees a greater increases for a bigger value of n than 3+15n, hence we can theoretically narrow down the time complexity of this entire program to O(n^2).

Therefore, complexity = O(n^2)

CONCLUSION

Hence, from this program, we have executed a complicated method of file reading by using the bizarre and complex elements as observed in an actual Rube Goldberg Machine.

Due to the complex nature of the system, we learned a lot about ADTs, data structures, APIs and other different methods of error rectification, input classification, data cleaning and file handling commands which are essential for software development on a daily basis.

One of the most important objectives that was achieved was learning the difference between an ADT and a data structure. Another major accomplishment was understanding file handling and data classification in the real world. Implementation of complicated algorithms like quicksort in a linked list was also achieved successfully.

This project taught us a lot about technical report-writing and coding in C++.

CONTRIBUTION

A total of four members contributed to the fruition of this project. All concepts were thoroughly shared, distributed, learned and executed by all four members.

Specific Contributions

Priyanka Srinivas (RA1911026010014)

1. Structured, wrote and formatted the report.
2. Provided the quickSort algorithm to sort data in ascending order.
3. Designed the function to take input to add to the file and check for correctness.
4. Implemented data cleaning on live user inputs.

Sarthak Gupta (RA1911026010041)

1. Designed an algorithm to read files.
2. Included comments and headers along with other members to improve clarity.
3. Implemented queue using linked list to distribute input read from the file.
4. Executed data cleaning from file.

Arpita Muleva (RA1911026010046)

1. Prepared and structured code to incorporate an unordered binary tree.
2. Created algorithm to shift data from queue to binary tree.
3. Wrote and implemented code to traverse binary tree in preorder and postorder methods.
4. Used inorder traversal to insert elements from unordered binary tree to linked list.

Shravya Sharan (RA1911026010055)

1. Designed a function to dequeue elements and print them in the correct order.
2. Implemented enqueue function without external libraries to generate a queue and store elements.
3. Used queue reversal by stack to display the elements in the required order.
4. Error rectification for linked list inputs.

BIBLIOGRAPHY

Geeks for Geeks for linked list - https://www.geeksforgeeks.org/data-structures/linked-list/

For quicksort - https://www.geeksforgeeks.org/quick-sort/ and https://www.programiz.com/dsa/quick-sort

StackOverflow for complexity - https://stackoverflow.com/questions/16917958/how-does-if-affect-complexity

Hacker Noon for Big O notation - https://hackernoon.com/big-o-for-beginners-622a64760e2

Techie Delight for queue - https://www.techiedelight.com/queue-implementation-cpp/

For .exe file - https://youtu.be/AvrjQtFBJvk