Course:

Data structure & Algorithm

Synthesis Report

Part 3

Applications

Guide teachers

Lê Ngọc Thành

Tạ Việt Phương

Phạm Trọng Nghĩa

Academic year

2020-2021

Thank you

Introduction members:

Full name	Student ID
Nguyễn Quốc Huy	20127518
Lại Minh Thông	20127635
Nguyễn Lê Hoàng Thông	20127078
Trần Anh Kiệt	20127545
Đào Đại Hải	20127016

Contents

APPLICATION 1: Buy movie tickets

I. Problem

- **A movie theater** wants a program to **sell tickets** and manage customer information who are members of them. They will keep customer information such as ID, name, and number of points that customers accumulate after purchasing tickets. These points can be exchanged for discounts.
- If the customer is not a member, we can register them as a member or not.
- Accumulated points can only be applied to customers who are already members
- Assumption that we have 3 types of ticket:

Price of ticket	**Point**
LOW: 60000đ	1
NORMAL: 80000đ	2
HIGH: 100000đ	3
10 points = 50000đ (discount)	

II. Solve problem

We will organize the program in a structured way including Person, Hashtable, Seat and Cinema.

1. **Person structure**
   This structure will keep customer information who are members of cinema including ID, name, and accumulated point. Notice that the customer's ID will be encoded to security.
2. **Hashtable**
   - This structure will store all customers as a hash table. The ID in the encrypted form will be hashed to become the key stored in the hashtable. For collision handling, we use double hashing in this problem.

- The hash table's data will be read and written to the file.

3. **Seat**

   Every seat in the movie theater has a price, status booked or not, and point that you will get after book it.

4. **Cinema**

   This is the biggest structure that you can manage hashtable and seat. Its role:

   - Reserve seat for customer: use array to store all seat of the cinema so we can apply **binary search** to find number seat for customer.
   - Cancel seat: cancel seat that customer has already chosen
   - Payment: make payments and accumulate points for cuscomer
   - Discount:
     - If the customer's accumulated points meet the criteria for a discount, the discount will be applied to the bill immediately
     - If the customer has just registered as a member and has enough accumulated points, the discount will also be applied.
   - Calculate sold tickets and revenue

APPLICATION 2: Grocery Store

I. Problem

- Create a **grocery store** sales app. In the store, we must manage and store product information such as **name**, **price**, and **quantity**.
- When importing and selling to customers, the application can adjust the quantity. We need to make a shopping cart for each customer so they can add and remove the products they want.
- We will proceed to pay the customer after they have selected the product.
- When a product is imported, we can adjust the price as well as the quantity on the store side.

II. Solve Problem

We will structure the program by using **stock items**, **stock lists**, **binary search trees**, **baskets** (shopping carts), and **linked lists**.

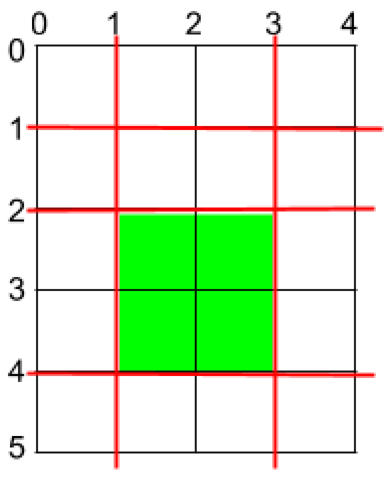
1. ***Stock item***
   Stock items will save product information such as name, price, quantity, and quantity in the cart but not yet paid for.
2. ***Stock list***
   A stocklist is a list that keeps track of all the products in the store. When the application is active, the stock list will read the product data from the file and save it back to the file when finished.
3. ***Binary Search Tree***
   To store products, we will use a binary search tree, which will speed up the search.
4. **Basket**
   The shopping cart will save the products the customer has chosen as well as the quantity; the customer can add or remove products. We will charge all of the products in the cart for the customer to pay once they have made a selection.
5. **Linked List**
   The products in the basket will be stored in a linked list. It is responsible for adding and removing items from the basket.

APPLICATION 3: Maximum Area

I. Problem

- *Maximum Area of a Piece of Cake After Horizontal and Vertical Cuts*.
- You are given a rectangular cake of size **h x w** and two arrays of integers horizontalCuts and verticalCuts where:
    - **horizontalCuts[i]** is the distance from the top of the rectangular cake to the ith horizontal cut and similarly, and
    - **verticalCuts[j]** is the distance from the left of the rectangular cake to the jth vertical cut.
    - Return the maximum area of a piece of cake after you cut at each *horizontal* and *vertical* position provided in the *arrays* **horizontalCuts** and **verticalCuts**.

For example:



Input: h = 5, w = 4, horizontalCuts = [1,2,4], verticalCuts = [1,3]

Output: 4

Explanation: The figure above represents the given rectangular cake. Red lines are the horizontal and vertical cuts. After you cut the cake, the green piece of cake has the maximum area.

II. Solve problem

We can find the max height and the max width separately. Our final answer will be **maxHeight** * **maxWidth**. Each height and width are defined by the distance between 2 cuts.

In the above example, the max height of 2 is defined by the distance between cuts 2 and 4 (4 - 2 = 2). To find all heights and widths, we must first **sort** our inputs **horizontalCuts** and **verticalCuts**. This will ensure that all of the cuts that are beside each other on the cake are also beside each other in the array.

Then, we can iterate through the sorted inputs one at a time and find each height or width by simply taking the difference between two adjacent cuts.

One thing to be careful about is the edges. For cuts in the middle, the distance is defined by the difference between two cuts. However, for the edges, they are defined by the cake's dimensions.

- The top-most cut's height will be equal to **horizontalCuts[0]**, while the bottom-most cut's height will be equal to
  h - horizontalCuts[horizontalCuts.length - 1].
- The left-most cut's width will be equal to **verticalCuts[0]**, while the right-most cut's width will be equal to
  w - verticalCuts[verticalCuts.length - 1].

Algorithm:

1. Sort both **horizontalCuts** and **verticalCuts** in ascending order (we will employ quick sort).

2. Initialize a variable **maxHeight** as the larger of the top and bottom edge:
   maxHeight = max(horizontalCuts[0],
   h - horizontalCuts[horizontalCuts.length - 1])

3. Iterate through **horizontalCuts** starting from index 1 (skip the 0th index since it represents the edge cut, which we accounted for in the previous step). At each iteration, find the height defined by the ith cut and the nearest cut above, **horizontalCuts[i] - horizontalCuts[i - 1]**. Update **maxHeight** if necessary.

4. Initialize a variable **maxWidth** as the larger of the left and right edge: **maxWidth = max(verticalCuts[0]**, **w - verticalCuts[verticalCuts.length - 1])**

5. Iterate through **verticalCuts** starting from index 1. At each iteration, find the width defined by the i^{th} cut and the nearest cut to the left, **verticalCuts[i] - verticalCuts[i - 1]**. Update **maxWidth** if necessary.

6. Our maximum area is **maxHeight * maxWidth**. Don't forget the modulo $10^9 + 7$, be careful of overflow. Return the maximum area.

APPLICATION 4: Book Of Address

I. Application: A program that works the same with the real-life Book of Phone Numbers. But the user might want to find a person's phone numbers by their last names or first names alternately. So, the program must allows the user to look up and visualize the list of people with the ascending order of their first or last names.

II. Step to implement:

- Read all the information of a person include: first name, last name, and phone numbers.
- Using any sorting algorithms to sort the list in alphabetical order. (In this demo, I will implement *Insertion Sort*)

III. Implementations:

```
struct Address
{
	string fname, lname;
	int num;
};
```

First, let's define a structure to contain a person's information.

```
void readFromFile(Address*& Book,int &n, string filename)
{
	ifstream fin(filename, ios::in);

	fin >> n; fin.ignore();
	Book = new Address[n];

	for(int i = 0;i < n;i++)
	{
		getline(fin, Book[i].fname, ' ');
		getline(fin, Book[i].lname, ' ');
		fin >> Book[i].num; fin.ignore();
	}

	fin.close();
}
```

Build a function to take the data by reading a file.

```
void listByFirstName(Address*& Book, int n)
{
	for (int i = 1; i < n; i++)
	{
		Address v = Book[i];
		int j = i - 1;
		while (j >= 0 && Book[j].fname > v.fname)
		{
			Book[j + 1] = Book[j];
			j--;
		}
		Book[j + 1] = v;
	}
}
```

Finally, Build a sorting function by applying *Insertion Sort* algorithms to sort the list by their first names.

```
void listByLastName(Address*& Book, int n)
{
	for (int i = 1; i < n; i++)
	{
		Address v = Book[i];
		int j = i - 1;
		while (j >= 0 && Book[j].lname > v.lname)
		{
			Book[j + 1] = Book[j];
			j--;
		}
		Book[j + 1] = v;
	}
}
```

You can also build another function to sort the list by their last names.

APPLICATION 5: First Bad Version

I. Problem: Suppose that you are a product manager and currently leading a team to develop a new product. But an error that is unfortunately occurred and lead to your product fails the quality check.

So now, you want to find the first version that made your product's quality went badly. Since each version is developed based on the previous versions, all the versions after a bad version are also bad.

- You are given an API *bool isBadVersion(version*) to check whether a version is bad or not and you should minimize the number of calls to the API.
- Implement a function to find the first bad version.

II. Solution: You could simply find the bad version by traversing sequencely by checking all the versions until you found the bad one. But this will be time consuming and is not a very effective way.

So, to take the advantage of the property of all the versions after the first bad one are also bad, you can apply the *Binary Search* algorithm to find the first bad version swiftly and reduce the number of times to call the API to check a version.

Implementation:

```
int firstBadVersion(int n) {
    int left = 1, right = n;
    int mid;
    while (left <= right)
    {
        mid = left + (right - left) / 2;
        if (isBadVersion(mid))
            right = mid - 1;
        else
            left = mid + 1;
    }

    return left;
}
```

APPLICATION 6: Mr. Ngau – Mrs. Ngau

I. Problem:

You probably know the annual "Mr. Ngâu, Mrs. Ngâu" day, which is a day full of rain and tears. However, the day before that, God's house allowed the reunion. In the galactic universe where Mr. Ngâu and Mrs. Ngâu reign, there are N planets numbered from 1 to N, he is on planet Adam (with number S) and she is on planet Eve (with number as T). . They need to find each other.

N planets are connected by a rainbow system. Any two planets can have no or only one rainbow (two-dimensional) connecting them. They always go to the goal by the shortest path. They travel at a constant speed and faster than the speed of light. Their meeting point could only be on a certain 3rd planet.

Requirements: Find a planet such that Mr. Ngâu and Mrs. Ngâu go there at the same time and the arrival time is the earliest. Know that two people can pass through the same planet if they arrive at that planet at different times.

- **Input:** The first line is 4 numbers N M S T ($N \leq 100$, $1 \leq S \neq T \leq N$), M is the rainbow number. The next M lines, each containing two numbers I J L, represent a rainbow connecting the two planets I , J and that rainbow has a length of L ($1 \leq I \neq J \leq N$, $0 < L \leq 200$).
- **Output:** Due to the nature of the rainbow, each year is different, so if there are no planets that meet the requirements, write a CRY line. If there are many planets that satisfy, write down the planet with the smallest index.

Example:

INPUT	OUTPUT

4 4 1 4
1 2 1
2 4 1
1 3 2
3 4 2
	2

II. Analysis:

+ Any two planets are only connected by at most one rainbow

+ Mr. Ngau and Mrs. Ngau always go to the goal in the shortest way

+ They travel at a constant speed and faster than the speed of light

In fact, this is a graph problem, we have the following algorithm:

-From planet S (where Mr. Ngau lives), we build a table man, where man[i] is the shortest path from planet S. Planet S to planet i (because Mr. Ngau always goes to his goal by the shortest path). man[i] = 0 mean there is no path from planet S to planet i.

-Similarly, we will build a table women, where women[i] is the shortest path from planet T to planet i. And women[i] = 0 mean there is no path from planet T to planet i.

-Due to the requirement of the problem is to find a planet other than S and T that 2 Ngau and his wife arrived at the same time and in the fastest time. That is, we will find the planet h such that (h is different from S and T) and (SP[h] = ST[h]) reaches the value minimum non-zero 0. If there is no such planet h, then we say CRY

-To build an array of SP and ST, we choose Dijkstra's algorithm to find the shortest path between two graph vertices.

-At the end of Dijkstra's algorithm, we will use the Linear search algorithm to find out if there is any point where two people meet and at the same time in that search we also find the smallest distance of the point where two people can meet.

```
void DIJKTRA(int N, vector < int > ke[1000], int a[][1000], int S, int d[])
{
    int MIN = 10000, u;
    bool F[1000];

    for (int i = 1; i <= N; i++)
    {
        d[i] = 10000;
        F[i] = 0;
    }

    d[S] = 1;

    while (true)
    {
        u = 0;
        for (int i = 1; i <= N; i++)
            if (F[i] == 0 and MIN > d[i])
            {
                MIN = d[i];
                u = i;
            }
        if (u == 0) break;
        F[u] = 1;
        for (int j = 0; j < ke[u].size(); j++)
        {
            int v = ke[u][j];
            if (F[v] == 0 && d[v] > d[u] + a[u][v])
                d[v] = d[u] + a[u][v];
        }
    }
}
```

```
int main()
{

    cin >> N >> M >> S >> T;
    for (int i = 1; i <= M; i++)
    {
        int x, y, t;
        cin >> x >> y >> t;
        //Because the graph is 2-dimensional,
        //we will when we use the vector to store the vertex i adjacent to j,
        //the vertex j is also adjacent to i
        ke[x].push_back(y);
        ke[y].push_back(x);
        a[x][y] = a[y][x] = t;//Store the distance from vertex i to j and
vice versa thus a 2-D graph
    }

    DIJKTRA(N, ke, a, S, man);
    DIJKTRA(N, ke, a, T, woman);

    for (int i = 1; i <= N; i++)
    {
        //If there are planets to meet and the distance is smaller than the
old distance
        //if there is a planet to meet before
        if (man[i] == woman[i] and MIN > man[i])
        {
            MIN = man[i];
            Luu = i;
        }

    }
    if (MIN != 10000) = > Can find
    else cout << "CRY"
}
```

*Algorithm used:

+Dijktra

+Linear search

APPLICATION 7: Monk and Island

I. Problem

The monk visited the land of the Islands. There are a total of N islands numbered from 1 to N . Several pairs of islands are connected by two-way bridges run on water. Đat hates going over these bridges because they require a lot of effort. He is stands on Island No. 1 and wants to go to Island No. Find the minimum number of bridges that he will must go through, if he takes the optimal path. Assume that there is always a path from vertex 1 to n.

- **Input:** The first line contains T . Number of test tests . The first line of each test set contains two integers N , M. Each of the M lines contains two integers X and Y , indicating that there is a bridge between Island X and Island Y.
- **Output:** Print the answer for each test case in a new line.

Example:

INPUT	OUTPUT

2
3 2
1 2
2 3
4 4
1 2
2 3
3 4
4 2
	2
	2

II. Analysis

We use BFS algorithm because here the requirement is to find the shortest path between 1 to N but if using Dijktra, it is too complicated and here if using Dijktra, we have to assign weight to each edge is 1, and if using BFS we won't need.

+ We use array d[i] to store the number of vertices from vertex 1 to node I with d[1]=1, so the final output will be d[n]-1, we have -1 because the number of edges is equal to the number of vertices minus 1.

+ In the BFS function, we will use the linear search algorithm to find out when we have found the vertex N and vice versa it will help us go to other vertices if we have not found the vertex adjacent to the vertex N.

```
void BFS(vector < int > ke[1000],int n,int m,int d[1000])
{
    queue < int > qu;
    qu.push(1);
    d[1]=1;
    while (!qu.empty())
    {
        int u=qu.front();
        qu.pop();
        for(int i=0;i<ke[u].size();i++)
        {
            int v=ke[u][i];
            if(d[v]==0)
            {
                d[v]=d[u]+1;
                qu.push(v);
            }
            if(v==n)
                return;
        }
    }
}
```

```
int main()
{
    int q;
    cin>>q;
    for(int i=0;i<q;i++)
    {
        int m,n,d[1000];
        bool F[1000];
        vector < int > ke[1000];
        memset(d, 0, sizeof(d));
        cin>>n>>m;
        for(int j=0;j<m;j++)
        {
            int x,y;
            cin>>x>>y;
            /*We use vector to store the adjacent edge of
vertex i as j and vice versa*/
            ke[x].push_back(y);
            ke[y].push_back(x);
        }
        BFS(ke,n,m,d);
        cout<<d[n]-1<<endl;
    }
    return 0;
}
```

***Algorithms used:**

+Dijktra

+Linear Search

APPLICATION 8: Bank

I. Problem:

Design a data structure to stimulate managing bank accounts. Each account consists of a unique username, the amount of money currently in the account, and a list of transactions performed in history. Each transaction consists of an id which is different for every transaction, the action performed (withdraw, deposit, ...), and the amount of money transferred.

Both accounts and transactions have a unique value, which are username and id, so the tree structures can be effective.

The number of accounts can be large and the list get updated less often than the transaction, so I chose B-tree for the job. The number of transactions is usually smaller, so I decided to use a binary tree instead. The transaction history is more likely to be updated than to be looked up, which is the reason I chose the red-black tree over AVL to store the history, as red-black tree provides faster insertion and removal operation than AVL (Source: https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/).

In short, I want to use the b-tree to manage accounts, each node in this b-tree also hold a root node of a red-black tree used to manage transaction history.

II. Implementation:

1. Structures:

My program defined six structures made out of each other.

Start with a structure to store a transaction. Currently, it can store the transaction id, the action performed, and the amount of money transferred.

```
struct trans
{
	string id;
	string action;
	int amount;
};
```

Using the transaction structure as data, the red-black tree node for transaction history is defined.

```
typedef struct historyNode
{
        trans data;
        bool color;
        historyNode *left;
        historyNode *right;
        historyNode* parent;
} H;
```

The historyList structure holds the root node of a red-black tree and the number of nodes in that tree, which is the number of transactions performed history of an account.

```
struct historyList
{
        H* root;
        int count;
};
```

The structure to store a profile consists of a username, the current amount of money in the account, and a historyList.

```
struct account
{
        string username;
        int amount;
        historyList *history;
};
```

Using the account structure as data, the b-tree node for account manage tree is defined.

```
typedef struct accountnode
{
        account data[WAY - 1];
        accountnode* child[WAY];
        int count;
        accountnode* parent;
        bool isLeaf;
} A;
```

Finally, the accountList structure, similar to the historyList, stores the root node of the account b-tree and the number of accounts in that tree.

```
struct accountList
{
	A* root;
	int count;
};
```

2.Stored file:

First I design the form of the file use to store all information. This form is apply for both the input file that the program will read at the beginning, and the output file of the save function.

- First line in the file is a single integer number M , the number of account currently stored in the file.

- The number of transactions of the ith account is N_i .

- Information of each account is written on 1 + N_i lines, start from the second line of the file.

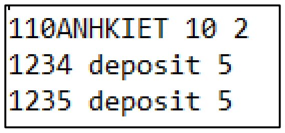
- The first line of each account will have form:

<username> <current money in account> <number of transaction in history (N_i)>

- The next N_i line will have form:

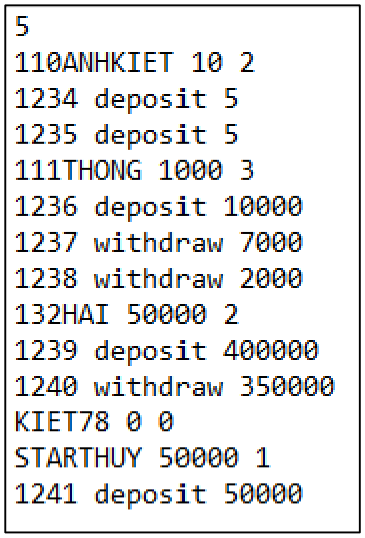
<id> <action> <amount of money>

Example for a single account stored form:



```
110ANHKIET 10 2
1234 deposit 5
1235 deposit 5
```

Example for the stored file:



```
5
110ANHKIET 10 2
1234 deposit 5
1235 deposit 5
111THONG 1000 3
1236 deposit 10000
1237 withdraw 7000
1238 withdraw 2000
132HAI 50000 2
1239 deposit 400000
1240 withdraw 350000
KIET78 0 0
STARTHUY 50000 1
1241 deposit 50000
```

The order of accounts and transactions in the input file may be random as long as they are in the right form. But in the output file of the save file function, the order of accounts is sorted in ascending by their username, and the transactions is sorted in ascending order by their id.

3.About source code:

The attached source code provide all operation functions on the b-tree and red-black tree (search, insert, delete).

The main program currently allow these action:

- Read file (done automatically).
- Search for profile with a given username.
- Search for specific transaction in profile with a given username and id.
- Create new empty profile (only have a username).
- Remove an existing profile with a given username.
- Save all information to a file.

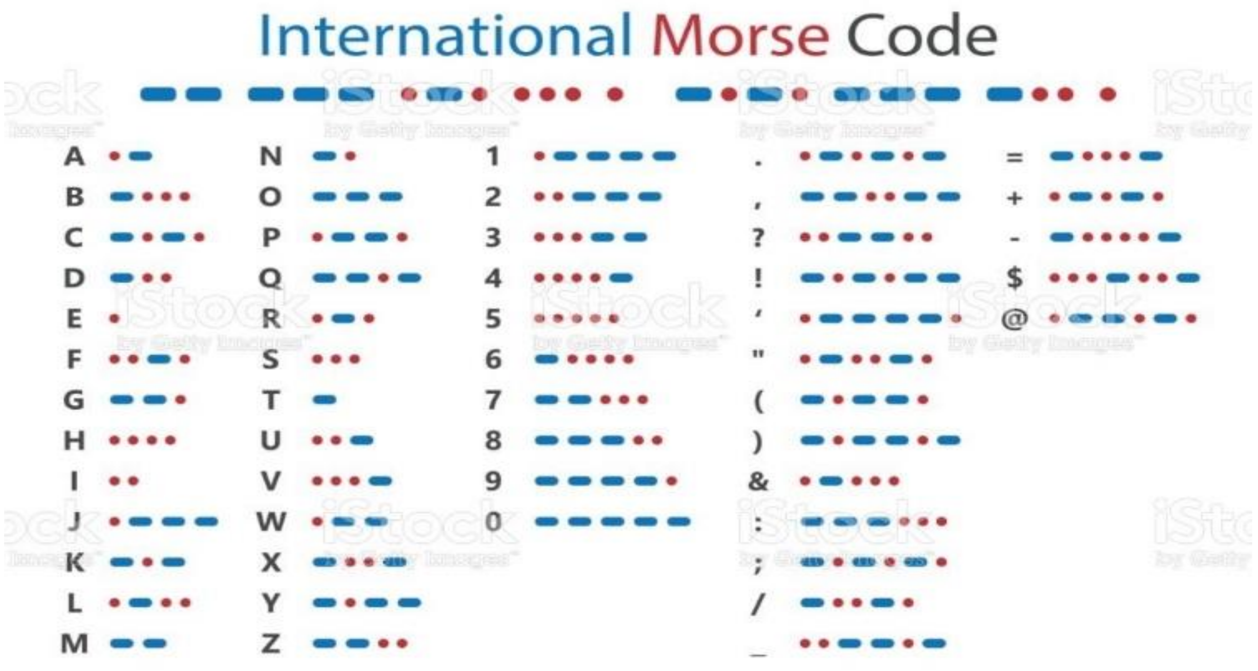
More action functions is still being work on.

APPLICATION 9: Morse code translator

I. Problem:

Design a simple program to translate the inputted morse code to a readable message. Use the International Morse Code table, the procedural signals are excluded.

Morse code is a method used in telecommunication to encode text characters as standardized sequences of two different signal durations, called dots and dashes, or dits and dahs. International Morse Code encodes the 26 Latin letters a through z, one non-Latin letter, the Arabic numerals, and a small set of punctuation and procedural signals (prosigns). There is no distinction between upper and lower case letters. (Source: https://en.wikipedia.org/wiki/Morse_code).



International Morse Code

-- --- .-. ... . -.-. --- -.. .

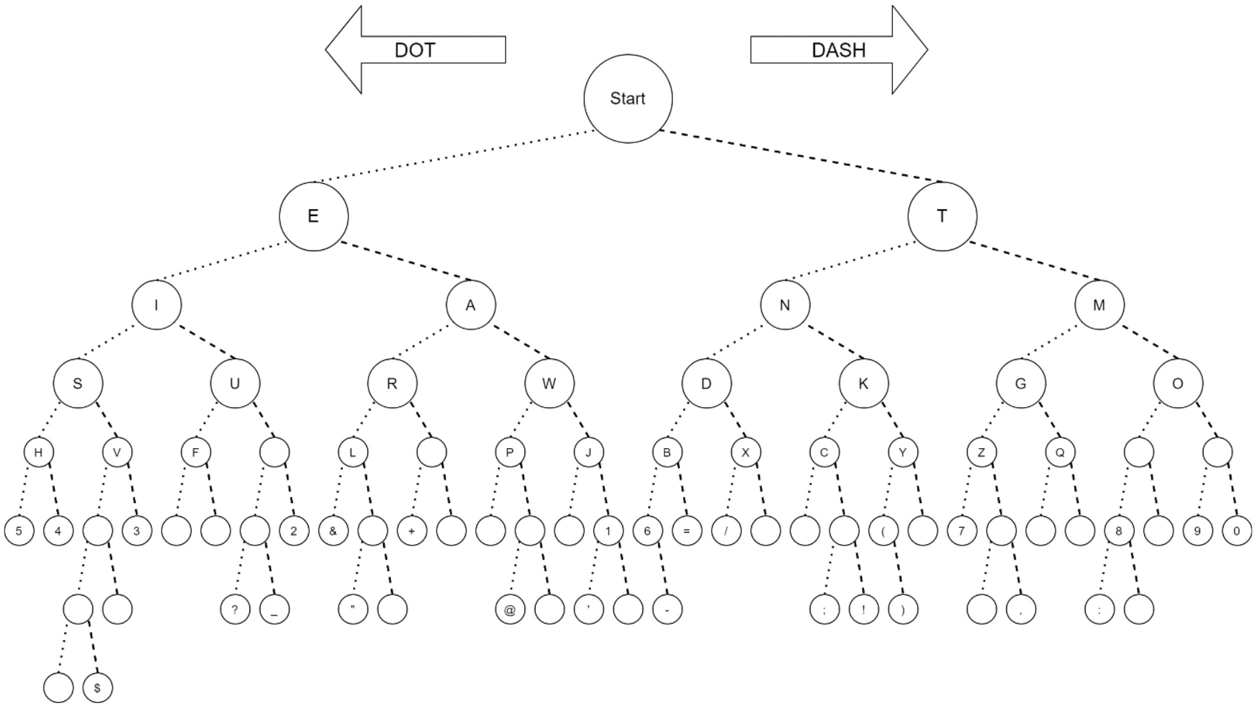
A	.-	N	-.	1	.----	.	.-.-.-	=	-...-
B	-...	O	---	2	..---	,	--..--	+	.-.-.
C	-.-.	P	.--.	3	...--	?	..--..	-	-....-
D	-..	Q	--.-	4	....-	!	-.-.--	$	...-..-
E	.	R	.-.	5	.....	'	.----.	@	.--.-.
F	..-.	S	...	6	-....	"	.-..-.		
G	--.	T	-	7	--...	(	-.--.		
H	....	U	..-	8	---..	)	-.--.-		
I	..	V	...-	9	----.	&	.-...		
J	.---	W	.--	0	-----	:	---...		
K	-.-	X	-..-			;	-.-.-.		
L	.-..	Y	-.--			/	-..-.		
M	--	Z	--..			_	..--.-		

There are many ways to store the decoded characters for each set of morse signal, among them, there is binary tree. Why ?

- Each character is encoded into a sequence of dots and dashes. This means each individual element of the encoded character only have two statuses: dot or dash. Similarly, the binary tree only has two possibilities in branching: left or right.

- The binary tree is only used as a storage of characters, and the number of these characters is fixed, or if changed in the future, it is still easy to re-organize the tree. Another benefit of this is that the tree as well as its height is constant as the program executing, no inserting, deleting, editing or balancing function needed.

The international morse code table can be organized as a binary tree if we use dots and dashes as the branch between nodes. The combination of branches on the path from root to the node holding a character is the morse encoded version of it. The tree look like this :



The drawback is this method does not prove effective in the reverse way, which mean translate from character to encoded morse code. The characters were not added to the tree by their value. Using the morse code to track a character is easy, but to track the path from begin to a character would require some extra stored information in each node or traversing the whole tree, which is not effective.

II. Implementation:

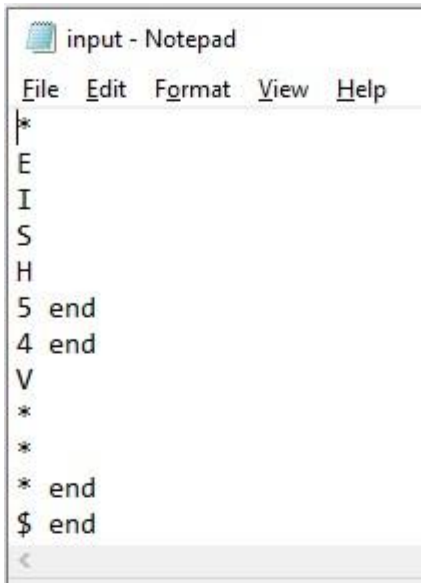
1. Input:

First, we must have an input file that stores the value for the tree. I want the file to have the pre-order form, the root is read first, then left child and right child. Each line in the file holds a single character which is the value that will be stored in the tree.

Lines holding values of the leaf nodes have an extra “end” follow the character to mark the end for the recursive call. The algorithm detects a leaf node when a line has more than one character, so “end” can be replaced with anything else, even a space.

The * character in the file mark value of an empty node in the tree, as * is nowhere to be found in the morse code table. If this change in the future, it can be fix easily.

The form of the stored input file:



```
*
E
I
S
H
5 end
4 end
V
*
*
* end
$ end
```

Now we go through form of the input morse code.

I use ‘.’ for dots and ‘-’ for dashes. For example, S is three dots “...” and O is three dashes “---”.

Morse code can be written in different ways. The two most common are shown below:

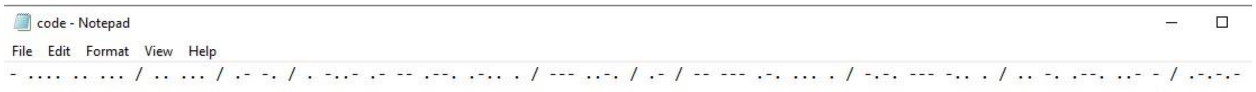
- Individual letters are separated by a space while individual words are separated by a forward slash (/).

- Individual letters are separated by a forward slash (/) and individual words are separated by a DOUBLE forward slash (//). Sentences in this format are separated by Four slashes (////).

(Source: https://www.geocaching.com/geocache/GC2T9M6_morse-code?guid=1490daee-7c43-4761-b8d7-be7db2d7b2d5)

For my program, I use the first way.

Example for an input sentence : - .... / .. ... / .- -. / . -..- .- -- .--. .-.. . / --- ..-. / .- / -- --- .-. ... . / -.-. --- -.. . / .. -. .--. ..- - / .-.-.-

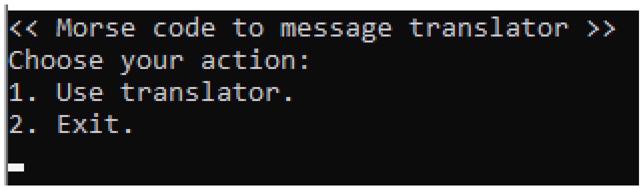


```
- .... .. ... / .. ... / .- -. / . -..- .- -- .--. .-.. . / --- ..-. / .- / -- --- .-. ... . / -.-. --- -.. . / .. -. .--. ..- - / .-.-.-
```

Note that for now, the input must be on a single line.

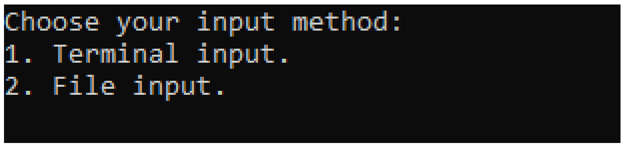
2. Program:

When run, the program bring up the main menu. Currently there are only two option to either use the program or exit.



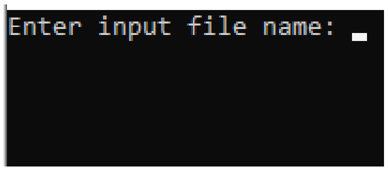
```
<< Morse code to message translator >>
Choose your action:
1. Use translator.
2. Exit.
```

To use the translator, user must first choose an input method :

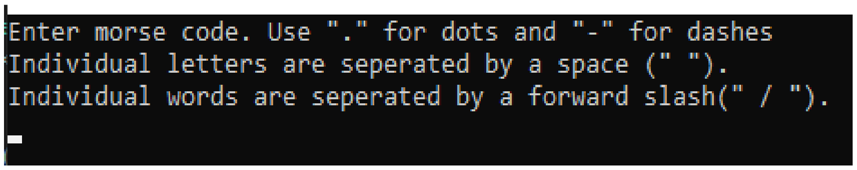


```
Choose your input method:
1. Terminal input.
2. File input.
```

Choosing input from file, the user will be asked for file's name, then the program will read input from that file. Otherway, the user will be guide to enter input form keyboard.

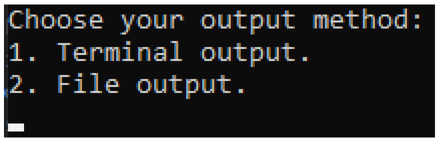


```
Enter input file name:
```



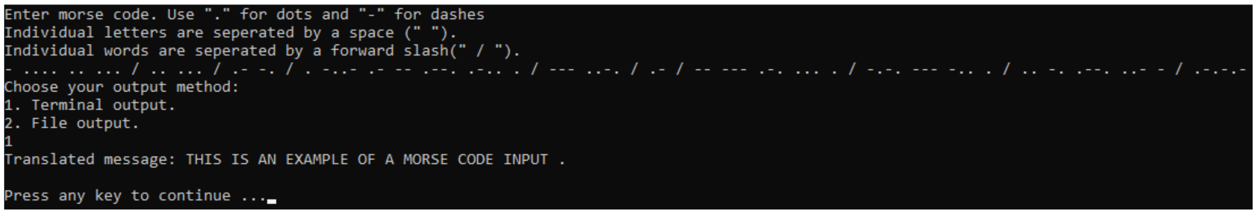
```
Enter morse code. Use "." for dots and "-" for dashes
Individual letters are seperated by a space (" ").
Individual words are seperated by a forward slash(" / ").
```

Afterward, the user is asked for an output method:



```
Choose your output method:
1. Terminal output.
2. File output.
```

Choosing output to file, user will need to enter the output file's name. Elsewise, the translated message will be display on terminal.



```
Enter morse code. Use "." for dots and "-" for dashes
Individual letters are seperated by a space (" ").
Individual words are seperated by a forward slash(" / ").
- .... .. ... / .. ... / .- -. / . -..- .- -- .--. .-.. . / --- ..-. / .- / -- --- .-. ... . / -.-. --- -.. . / .. -. .--. ..- - / .-.-.-
Choose your output method:
1. Terminal output.
2. File output.
1
Translated message: THIS IS AN EXAMPLE OF A MORSE CODE INPUT .

Press any key to continue ...
```

As the user press any key after this, the main menu will come back. This repeat until user choose to exit.

III. Source code:

```
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <conio.h>

using namespace std;

struct node
{
	char value;
	node* dot;
	node* dash;
};
```

```
node* createNode()
{
        node* newnode = new node;
        newnode->value = '\0';
        newnode->dot = NULL; // left child
        newnode->dash = NULL; // right child
        return newnode;
}

void readTree(node*& curr, ifstream& ifs)
{
        // read the input for the tree
        if (ifs.eof()) return;

        string s;
        getline(ifs, s);

        curr = createNode();
        // empty nodes keep value '\0'
        if(s[0] != '*') curr->value = s[0];

        // end of a path
        if(s.length() > 1) return;

        // read left child
        readTree(curr->dot,ifs);
        //read right child
        readTree(curr->dash, ifs);
}

char singleTranslate(node* root, string code)
{
        // translate a single character
        // if it is a slash, return a space character
        if (code == "/") return ' ';
        int index = 0;

        node* temp = root;
        while (index < code.length())
        {
                // if reach NULL node, break
                if (temp == NULL) break;
                // if any character in the source morse code is not dot or dash, break
                if ((code[index] != '.') && (code[index] != '-'))
                {
                        temp = NULL;
                        break;
                }
                // if the character is dot, go to dot subtree and vice versa
                if (code[index] == '.') temp = temp->dot;
                else temp = temp->dash;
                index++;
        }
        // if the input is violated, return NULL character
        if (temp == NULL) return '\0';
        return temp->value;
}

string messageTranslate(node* root, string code)
{
        // translate 1 line of morse code
        stringstream ss(code);
        string character, message = "";
        getline(ss, character, ' ');
        do
        {
                char translatedChar = singleTranslate(root, character);
                if (translatedChar == '\0') message += "<unable to translate>";
                else message.push_back(translatedChar);
                getline(ss, character,' ');
        } while (ss);
        return message;
}

string terminal_input()
{
        system("CLS");
        cin.ignore();
        string code;
        cout << "Enter morse code. Use \".\" for dots and \"-\" for dashes \nIndividual letters are
seperated by a space (\" \").\nIndividual words are seperated by a forward slash(\" / \").\n";
        getline(cin, code);
        return code;
}

string file_input()
{
        system("CLS");
        cin.ignore();
        string filename;
        cout << "Enter input file name: ";
```

```
		getline(cin, filename);
		ifstream ifs(filename);
		if (ifs)
		{
			string code;
			getline(ifs, code);
			return code;
		}
		return "\0";
}

void terminal_output(string mess)
{
		cout << "Translated message: " << mess;
}

void file_output(string mess)
{
		string filename;
		cout << "Enter output file name: ";
		getline(cin, filename);
		ofstream ofs(filename);
		ofs << mess;
}

void menu()
{
		node* root = NULL;
		ifstream ifs("input.txt");
		readTree(root, ifs);
		if (ifs)
		{
			while (1)
			{
				// choose to use or exit
				char choice = '\0';
				while ((choice != '2') && (choice != '1'))
				{
					system("CLS");
					cout << "<< Morse code to message translator >>\n";
					cout << "Choose your action: \n1. Use translator.\n2. Exit.\n";
					cin >> choice;
				}

				if (choice == '2') break;
```

```
                        string code, mess;

                        // choose input method
                        choice = '\0';
                        while ((choice != '2') && (choice != '1'))
                        {
                                system("CLS");
                                cout << "Choose your input method:\n";
                                cout << "1. Terminal input.\n2. File input.\n";
                                cin >> choice;
                        }

                        // read input
                        if (choice == '1') code = terminal_input();
                        else code = file_input();

                        if (code == "\0")
                        {
                                // break if the input file is not found
                                cout << "Input code file not found!";
                        }
                        else
                        {
                                // translate input
                                mess = messageTranslate(root, code);

                                // choose output method
                                choice = '\0';
                                while ((choice != '2') && (choice != '1'))
                                {
                                        cout << "Choose your output method:\n";
                                        cout << "1. Terminal output.\n2. File output.\n";
                                        cin >> choice;
                                }

                                if (choice == '1') terminal_output(mess);
                                else file_output(mess);
                        }

                        cout << "\n\nPress any key to continue ...";
                        _getch();
                }
        }
        // does not execute without the input stored file
        else cout << "Input file is missing!";
}
```

APPLICATION 10: Google Pagerank Algorithm

I. Introduction

PageRank is one the of oldest and most important algorithms of Google. Google Search uses it to rank web pages in its search engine. PageRank is created by Larry Page and Sergey Brin, founders of Google. PageRank is a way of measuring the importance of each website pages on the internet. Thanks to the superiority of PageRank and others algorithms, Google has beated competitors such as Yahoo or Bing to become the word's best search engine.

According to Google: "PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites." (Source: Wikipedia – PageRank).

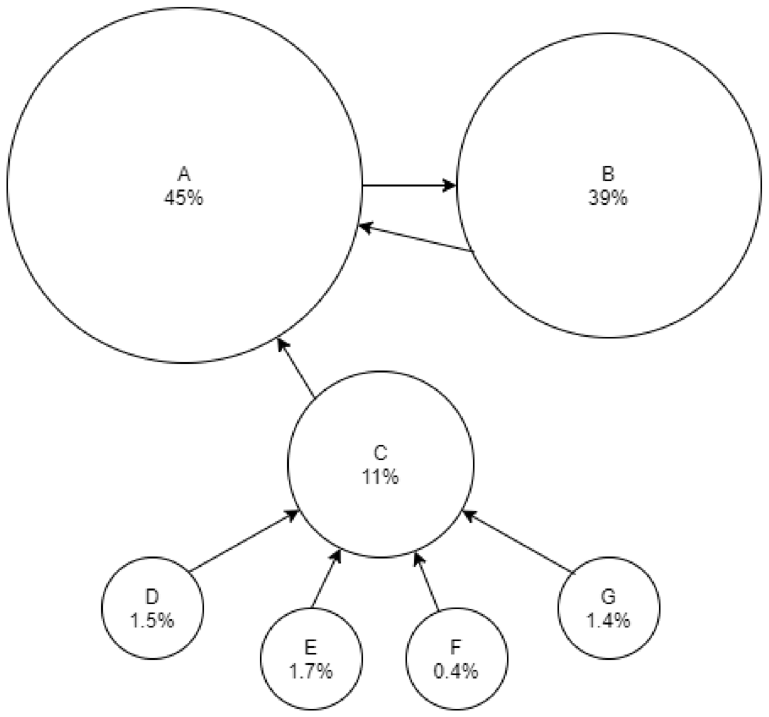
This algorithm is very complicated to the one who just study data structure for a little time, but it's a good application for Graph data structure implement in real life. So in this section, we will just clarify some basics about PageRank algorithms which directly relate to Graph data structure.

II. Algorithm implementation

PageRank value (sympol: PR(E)) is calculated by webgraph – Graph with the vertex is the website and the edge is the link to the website (these websites include some government website .gov and some authority else). Each link to a website will be counted as a increasing of value PageRank of that website. So, a website which has many links from other high PR(E) value website will have a high PR(E) value too, if a website just has some small link's PR(E) value link to it, its PR(E) value will not sure to be high).

PageRank also represents for probability that a user randomly clicks on the link on the internet websites. In PageRank, a website which PR(E) = 0.5 means: 50% chance of a user clicking on a random link to go to that website (of course in real life this percent hard to be happen).

Example of website's PR(E):



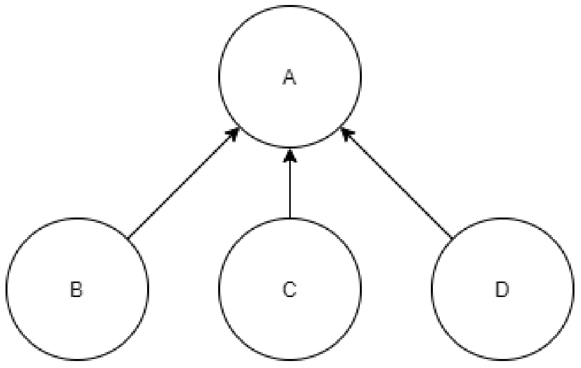
In this example, I use value percent for easy implementation. Website C has 11% that user will randomly click to the link which go to it, website B has 39%. Because website B just has one link but that link is from a important website (website A – 45%) so PR(E) of website B is very high. Website C has many links but its links are not very important with low PR(E).

III. Algorithm formula

Now we will prove the formula for the PageRank algorithm.

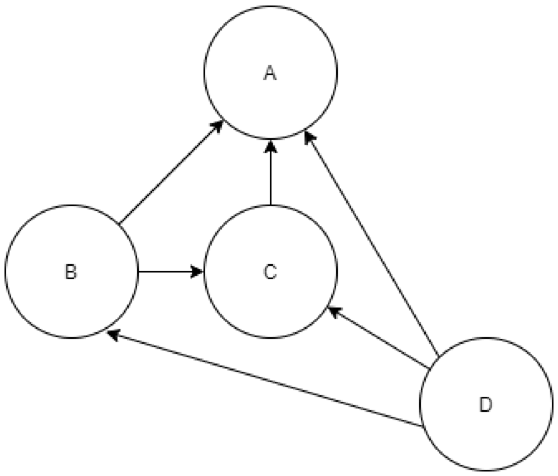
Assume that on the internet has four website: A, B, C, D. So at the beginning, no links are implement so the percent for each website is 0.25 (25%).

Example 1:



When website B, C, D have a link to website A, each website will tranfers its PR(E) to website A. So, $PR(A) = PR(B) + PR(C) + PR(D)$.

Example 2:



Now website B has another link to website C, website D has another links to website C and B. Website B must divide its PR(E) by 2 to tranfers its value to two website C and A. Website D must divide its PR(E) by 3 to tranfers its value to website A, B and C. So PR(A) now:

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}$$

In general, we have the formula for PageRank value to any website U:

$$PR(U) = \sum_{v \in B(U)} \frac{PR(v)}{L(v)}$$

v is the website that has a link to website U. B(U) is the set of many v. L(v) is the number of links that goes from website v.

IV. Algorithm in coding

According to the formula above, we can turn it to code. To calculate the PageRank value of a website: In a directed graph, detect the root vertex that stand for the website. Then we must to find out each vertex that adjacent to root vertex and number of edges begin of that vertex. We can use BFS algorithm to go to all adjacent vertices to vertex root, but remember to stop when the root vertex has no more unvisited predecessor.

Pseudo-code:

```
int adjEdgeNum(graph G, node V)
{
    sumEdge = 0
    For all edges of V
    {
        If edge starts from V: sumEdge++
    }
    Return sumEdge
}

int PRE(graph G, node W) // W is root website vertex
{
    Sum = 0
    For all adjacent vertices to W (adj[W][i])
    {
        Sum += adj[W][i] / adjEdgeNum(adj[W][i])
    }
    return Sum
}
```