Data Structure

Hands on Lab

March 2011

For the latest information, please see bluejack.binus.ac.id
Information in this document, including URL and other Internet Web site references, is subject to change without notice. This document supports a preliminary release of software that may be changed substantially prior to final commercial release, and is the proprietary information of Binus University.

This document is for informational purposes only. BINUS UNIVERSITY MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

The entire risk of the use or the results from the use of this document remains with the user. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Binus University.

Binus University may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Binus University, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2011 Binus University. All rights reserved.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.
Table of Contents

OVERVIEW ................................................................................................................................. iii

SYSTEM REQUIREMENT...................................................... Error! Bookmark not defined.

Chapter 01 – Struct................................................................. Error! Bookmark not defined.

Chapter 02 – Stack............................................................... Error! Bookmark not defined.

Chapter 03 – Queue .................................................................................................................. 18

Chapter 04 – Single Linked List................................................................. 28

Chapter 05 – Double Linked List............................................. Error! Bookmark not defined.

Chapter 06 – Binary Tree ..................................................................................................... 51

Chapter 07 – Binary Search Tree................................................................. 65

Chapter 08 – Binary Search Tree (Parent Node).......................... 81

Chapter 09 – AVL Tree ........................................................... 102

Chapter 10 – HEAP .................................................................................. 112
OVERVIEW

Chapter 01
• Introduction of Struct

Chapter 02
• Using Struct to Create Stack

Chapter 03
• Using Struct to Create Queue

Chapter 04
• Single Linked List : Insert and Delete Data

Chapter 05
• Double Linked List : Insert and Delete Data

Chapter 06
• Binary Tree : Insert and Delete Data

Chapter 07
• Binary Search Tree : Insert, Update and Delete Data
• Traversal : Preoder, Inorder and Postorder

Chapter 08
• Binary Search Tree with Parent Node : Insert, Update and Delete Data
• Traversal : Preoder, Inorder and Postorder

Chapter 09
• AVL Tree : Insert and Delete Root

Chapter 10
• HEAP : Insert Data and Delete Root
SYSTEM REQUIREMENT

- **Hardware:**
  - Minimum:
    - 1.6 GHz CPU, 192 MB RAM, 1024x768 display, 5400 RPM hard disk
  - Recommended:
    - 2.2 GHz, 384 MB, 1280x1024 display, 7200 RPM or higher.
  - On Windows Vista:
    - 2.4 GHz CPU, 768 MB RAM

- **Software:**
  - Microsoft Visual Studio 2008 Express Edition (Visual C++)
  - 1.3 GB of available disk space for the full install
Chapter 01

Struct
1.1. Definition of Struct

**Struct** in C programming language is a structured type that aggregates a fixed set of labelled objects (variable), possibly of different types, into a single object.

A struct declaration consists of a list of variable, either one variable or more depending on the needs of the use of Struct.

**Total storage required** for a struct depends on the amount of total storage requirement of each variable.

1.2. How to use Struct

Example:

To declare a struct, is simply to use keyword “struct” and followed by declaring the variable.

```c
struct students
{
    char name [50];
    int age;
    float height;
}
```

To create a new variable of this type struct, we can simply to declare:

```c
struct students Budi;
```

In this case, we have a variable declared as “Budi”.

This variable has four properties, one char data type variable named “name”, one int data type variable named “age”, and two float data type variable named “height” and “weight”.

To access these four values we can simply use:

```c
strcpy(Budi.name, “Budi”);
Budi.age = 20;
Budi.height = 170.25;
```
1.3. Example

1.3.1. Example 01 – BlueDigi

BlueDigi Corporate is a company engaged in programming. Once, they want to make a program named Digipet. Digipet is a program to raise and take care of the virtual pets. BlueDigi hired you to make the prototype of the program.

Here are the descriptions of the prototype:

- The program has two (2) pets to take care. The program always shows the pets’ status (Happiness, Health, and cleanliness).
  
  Each status has 10 points at the beginning of the program started.

- The program consists of 4 menus:
  
  ✓ 1. Play
  ✓ 2. Eat
  ✓ 3. Bath
  ✓ 4. Exit

- If user chooses “Play” (menu 1), then the program will:
  
  ✓ Ask user to choose which pet that he/she wants to play with
  ✓ Playing will affect the status of selected pet such as:
    
    - Happiness will increase 3 points
    - Health will decrease 1 point
    - Cleanness will decrease 1 point

- If user chooses “Eat” (menu 2), then the program will:
  
  ✓ Ask user to choose which pet that he/she wants to feed
  ✓ Eating will affect the status of selected pet such as:
    
    - Happiness will decrease 1 point
    - Health will increase 3 points
    - Cleanness will decrease 1 point

- If user chooses “Bath” (menu 3), then:
  
  ✓ Ask user to choose which pet that user wants
  ✓ Bath will affect the status of selected pet such as:
    
    - Happiness will decrease 1 point
    - Health will decrease 1 point
- Cleanness will increase 3 points.

- If user chooses “Exit” (menu 4), then the program will ends

1.3.2. Explanation

1. Create Project Visual C++ in Microsoft Visual C++ Express 2008
   a. Run Microsoft Visual C++ Express 2008 from “Start Menu”
   b. Open Menu File → New → Project
   c. On Project Window, choose Visual C++ on tree panel on the left side window
   d. Choose “Win32 Console Application” on middle panel of project window
   e. Enter name solution “Soal01.sln”
   f. Browse location of solution in D:\
   g. Then press OK button

2. Create the program
   a. Go to main.cpp on “Solution Explorer” and double click on the file “Soal01”
   b. Type include stdio.h on main.cpp

```cpp
#include <stdio.h>
```

b. Type include stdio.h on main.cpp

c. Declare struct data as digipet data which has properties on following:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td>Integer</td>
<td>Number of digipet</td>
</tr>
<tr>
<td>Happiness</td>
<td>Integer</td>
<td>Level of digipet Happiness</td>
</tr>
<tr>
<td>Health</td>
<td>Integer</td>
<td>Level of digipet health</td>
</tr>
<tr>
<td>Cleanness</td>
<td>Integer</td>
<td>Level of digipet Cleanness</td>
</tr>
</tbody>
</table>
3. Create Play Method

Play method will allow user to play with selected digipet.
If digipet is playing, then its Happiness level is increased by 3 point from current, but it will decrease Health level and Cleanness level by 1 point.
This method take one parameter struct data and passing by reference.

```c
void play(struct data &digi){
    digi.happiness += 3;
    digi.health -= 1;
    digi.cleaness -= 1;
}
```

4. Create Eat Method

Eat method will allow user to feed selected digipet.
If digipet is eating, Health level is increased by 3 points, but it will reduce Happiness and Cleanness level by 1 point.
This method take one parameter struct data and passing by reference.

```c
void eat(struct data &digi){
    digi.health += 3;
    digi.cleaness -= 1;
    digi.happiness -= 1;
}
```

5. Create Bath Method

Bath method will allow user to bathe selected digipet.
If digipet is taking bath, Cleanness level is increased by 3 points, but it will reduce Happiness and Health level by 1 point.
This method take one parameter struct data and passing by reference.

```c
void bath(struct data &digi){
    digi.cleaness += 3;
    digi.health -= 1;
    digi.happiness -= 1;
}
```
6. Create Status Method

Status method will give user information about current condition and properties of selected digipet.

This method take one parameter struct data and passing by reference

```c
void status(struct data digi)
{
    printf("%c%Digipet %d's Status\n",3,digi.num);
    for(int i=0;i<20;i++)
        printf("%c",205);
    printf("\nHappiness\n:t : %d\n",digi.happiness);
    printf("Health\n:t : %d\n",digi.health);
    printf("Cleanness\n:t : %d\n",digi.cleaness);
}
```

7. Create Cek Method

“Cek” method will give user suggestion information what user must do with digipet whether digipet is sad, sick, or dirty.

This method take one parameter struct data and passing by reference.

```c
void cek(struct data &digi){
}
```

a. Check on Happiness properties, if Happiness less or equals zero then system will print “Your Digipet <<number>> is going to sad, play with it..”.

Number of digipet is from num properties

```c
if(digi.happiness <= 0){
    digi.happiness = 0;
    printf("Your Digipet %d is going to sad, play with it..",digi.num);
    getchar();
}
```

b. Check on Health properties, if Health less or equals zero then system will print “Your Digipet <<number>> is going to sick, it must eat..”.

Number of digipet is from num properties

```c
if(digi.health <= 0){
    digi.health = 0;
    printf("Your Digipet %d is going to sick, it must eat..",digi.num);
    getchar();
}
```
c. Check on Cleaness properties, if Cleanness less or equals zero then system will print “Your Digipet <<number>> is going to sick, it must eat..”.

Number of digipet is from num properties

```c
if(digi.cleaness <= 0){
    digi.cleaness = 0;
    printf("Your Digipet %d is going to dirty, it must go to bath..",digi.num);
    getchar();
}
```

d. At the end of this method, all properties from digipet will be checked. If each property more than 100, then property value will be 100.

```c
if(digi.happiness >= 100)
    digi.happiness = 100;
if(digi.health >= 100)
    digi.health = 100;
if(digi.cleaness >= 100)
    digi.cleaness = 100;
```

8. Create Choose Method

Choose method allow user to choose which digipet user want to play. There is a validating process with input value between 1 and 2.

This function will return input value to its caller method.

```c
int choose(){
    int pet;
    do{
        printf("Choose your digipet [1..2]: ");
        scanf("%d",&pet);
        fflush(stdin);
    }while(pet!= 1 & & pet !=2);
    return pet;
}
```

9. Create Clear Method

Clear method will clear the screen with 25 lines of enter character (“\n”).

```c
void clear(){
    for(int i=0;i<25;i++)
        printf("\n");
}
```
10. Main Method

a. Declare two digipet variables from struct data.
   First variable named `digi1`, and second variable named `digi2`.

```c
struct data digi1;
struct data digi2;
```

b. Initialization two digipet. Set num property value from `digi1` to 1 and set num property value from `digi2` to 2 and the rest of properties values is 10

```c
digi1.num = 1;
digi1.happiness = 10;
digi1.health = 10;
digi1.cleaness = 10;

digi2.num = 2;
digi2.happiness = 10;
digi2.health = 10;
digi2.cleaness = 10;
```

c. Declare one variable “pil” to collect user input menu choosen, called “pil” with data type integer

```c
int pil;
```

d. Create a main process loop in system. In this part, we will create a main loop process which will loop until user give value 4 into “pil” variable.

```c
do{
}
while(pil!=4);
```

e. Print Digipet Title surrounded with rectangle. Print rectangle 9x9 character with for-loop. Use ASCII character 186, 187, 188, 200, 205 to print the rectangle.

```c
clear();
pref("\t%c",201);
for(int i=0;i<9;i++)
    printf("%c",205);
pref("%c\n",187);
pref("\t%c Digipet %c\n",186,186);
pref("\t%c",200);
for(int i=0;i<9;i++)
    printf("%c",205);
pref("%c",188);
```
f. Print both digipet status with status method

```c
printf("\n\n");
status(digi1);
printf("\n\n");
status(digi2);
printf("\n\n");
```

g. Print menus that user can choose

```c
printf("Choose Digipet Activity :\n");
printf("1. Play\n");
printf("2. Eat\n");
printf("3. Bath\n");
printf("4. Exit\n");
```

h. Create input menu to allow program captures user input menu from screen

```c
printf("Input your choice : ");
scanf("%d", &pil);
fflush(stdin);
```

i. Create option menu from user’s choice with switch case syntax with condition:

- If user chooses first menu with value of variable “pil” is 1, then program will run “choose” method to ask user which digipet user wants to play, and then the result from “choose” method will be used for selecting between digi1 or digi2 to play in main method.
- If user chooses first menu with value of variable “pil” is 2 then program will run “choose” method to ask user which digipet user wants to feed, and then the result from “choose” method will be used for selecting between digi1 or digi2 to feed in main method.
- If user chooses first menu with value of variable “pil” is 3 then program will run “choose” method to ask user which digipet user wants to bathe, and then the result from “choose” method will be used for selecting between digi1 or digi2 to bathe in main method.
Check digipet current condition. Program will check up both digipet and tell user what user must do to both digipet with “cek” method

```java
cek(digi1);
cek(digi2);
```

1.4. Exercise

BLUE Cake Shop is a small cake shop. Mr. Krauser, the manager of BLUE Cake Shop, wants to change their cashier system to be paperless. He needs a program that can handle his cake shop transaction. He asks you as a skillful programmer to make the program with these descriptions:

- Program always shows List of Cakes which contains **No**, **Cake Code**, **Cake Name**, **Available**, and **Price** of the available cakes. The List of Cakes:

<table>
<thead>
<tr>
<th>No.</th>
<th>Cake Code</th>
<th>Cake Name</th>
<th>Available</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>01.</td>
<td>CK001</td>
<td>Blueberry Cake</td>
<td>13</td>
<td>Rp. 25000,-</td>
</tr>
<tr>
<td>02.</td>
<td>CK009</td>
<td>Chocochip Cake</td>
<td>5</td>
<td>Rp. 20000,-</td>
</tr>
<tr>
<td>03.</td>
<td>CK017</td>
<td>Mayonaise Cake</td>
<td>24</td>
<td>Rp. 30000,-</td>
</tr>
<tr>
<td>04.</td>
<td>CK023</td>
<td>Strawberry ShortCake</td>
<td>7</td>
<td>Rp. 17500,-</td>
</tr>
</tbody>
</table>
• Program consists of 3 menus:
  1. Sell
  2. Add Stock
  3. Exit
• If user chooses Sell, then:
  o Ask user to input “cake code” he/she wants to sell.
  o The “cake code” must consist of 5 characters and be available in the list of cakes. If user inputs the “cake code” other than the ones in the list, the program will show the message “--- The Cake Code doesn’t exist ---” and ask user to input again. The “cake code” is case sensitive.
  o Then ask user to input the quantity that he/she wants to sell.
  o The quantity must be between 0 and x, where x = [the quantity available of chosen cake].
  o The user cannot sell the cakes more than the quantity available on the list. If user tries to do so, show the message “...The quantity of cake is not enough...”
  o If user succeeds to sell, then show message:
    “Total Price is: Rp [price of cake] - x [quantity of cake] = Rp [total price],”
    “--- Thank You ---”
  o Then subtract the available cakes with the quantity of cake that has been sold.
    [available cakes] = [available cakes] - [quantity of cakes that has been sold]
• If user chooses Add Stock, then:
  o Ask user to input the “cake code” he/she wants to add to the list.
  o The “cake code” must consist of 5 characters and be available in the list of cakes. If user inputs the “cake code” other than the ones in the list, program will show message “--- The Cake Code doesn’t exist ---” and ask user to input again. The “cake code” is case sensitive.
  o Then ask user to input the quantity.
  o Validate the quantity of cake must be between 1 and 10.
  o If user succeeds to add stock, then show message
    “--- Adding Stock Success ---”
Then add the available cakes with the quantity of cake that has been added.

\[
\text{[available cakes]} = \text{[available cakes]} + \text{[quantity of cakes added]}
\]

- If user chooses Exit, then the program ends.

Print Screen of Main Menu

<table>
<thead>
<tr>
<th>BLUE CAKE SHOP CASHIER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
<td>Cake Code</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
</tr>
<tr>
<td>01.</td>
<td>CK001</td>
</tr>
<tr>
<td>02.</td>
<td>CK009</td>
</tr>
<tr>
<td>03.</td>
<td>CK017</td>
</tr>
<tr>
<td>04.</td>
<td>CK023</td>
</tr>
</tbody>
</table>

Menu :
1. Sell
2. Add Stock
3. Exit
Input choice :

Print Screen of Sell Menu (Menu ‘1’)

Input Cake Code [5 chars]: CK009
Input Quantity [0..5]: 3

Total Price is : Rp 20000,- x 3 = Rp 60000,-

--- Thank You ---

Print Screen of Add Stock (Menu ‘2’)

Input Cake Code [5 chars]: CK023
Input Quantity [1..10]: 7

--- Adding Stock Success ---
Chapter 02

Stack
2.1 Definition of Stack
A stack is a last in first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop.

2.2 Push and Pop Operation

2.2.1 Push Operation
Push operation is an operation to add an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty.

2.2.2 Pop Operation
Pop operation is an operation to remove an item from the top of the stack and returns this value to the caller.
A pop either reveals previously concealed items, or results in an empty stack.
2.3. Example

2.3.1. Example 02 - Blue Rice Stockist

Mr. Ali is a seller of daily needs. Due to his rice stock system is not too good, he asks you to make a program using a stack concept with a concept of array of struct. Here are the descriptions of the program:

- Program always shows rice stock in stack.
- Program consists of 3 menus:
  - 1. Stock Rice Sack
  - 2. Sell Rice Sack
  - 3. Exit
- If user chooses "Stock Rice Sack" (menu 1), then:
  - Ask user to input type of rice then validate the type of the rice that must be 'long', 'medium', or 'short' with case sensitive. After that add the type of the rice with 'grain' automatically.
  - Ask user to input weight of rice sack then validate the weight that must be between 10 and 100 kilograms.
  - Maximum data that can be stored in the rice stack is only 10 then validate if the data has reached 10 show message “--- The Rice Storage is Full ---”
  - If data has been successfully added to the stack, then show message “--- Add Rice Sack Success ---”
- If user chooses Sell Rice Sack (menu 2), then:
  - If the stack is empty, then show message “--- The Rice Storage is Empty ---”
  - If the stack is not empty, then the program will remove an item from the top of the stack and show message “--- Sell Rice Sack Success ---”
  - If user chooses Exit, then remove all data on the stack and the program will ends.
2.3.2. **Explanation**

1. Create Solution “Answer02.sln” and save the solution in D:\

2. Declare array of struct to store rice type and weight. Set the value of array to maximum data number (10):

3. Go to main.cpp on Solution Explorer and double click on the file

![Solution 'Answer02' (1 project)

*Header Files

*Resource Files

*Source Files

[C:] Answer02.cpp]

4. Type include stdio.h, string.h, and stdlib.h

```c
#include <stdio.h>
#include <string.h>
```

5. Create a variable named **total** and initialize it with 0

```c
int total = 0;
```

6. Declare struct data for rice stack with following properties:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>char[50]</td>
<td>Rice Type</td>
</tr>
<tr>
<td>Weight</td>
<td>int</td>
<td>Rice Weight</td>
</tr>
</tbody>
</table>

```c
struct data
{
    char type[50];
    int weight;
}stack[10];
```

7. Create View Method

View method will show all items/data in the stack (rice type and weight).
Besides, it will show the top of the stack with an arrow.
Total is a variable to determine which data is on top of the stack.
8. Create Stock Method (push)

Stock method will allow user to input rice type and weight and then add it to the top of the stack. It will also check whether the stack is full or not (10 data maximum)

1. Checking if the stack is full.

```c
void stock()
{
    if (total<10)
    {
        //validating rice type (2)
        //validating rice weight (3)
        //validating the number of data (total) (4)
    }
    else
    {
        printf("\n\n\n --- The Rice Storage is Full ---\n");
    }
    getchar();
    fflush(stdin);
}
```

2. Validating rice type (rice type must be between long, medium, and short)

Method strcmp needs an additional library: string.h

```c
do
{
    printf(" Input Rice Type [long/medium/short] grain: ");
    scanf("%[^\n]", stack[total].type);
    fflush(stdin);
} while (strcmp(stack[total].type, "long")!=0 && 
        strcmp(stack[total].type, "medium")!=0 && 
        strcmp(stack[total].type, "short")!=0); 
strcat(stack[total].type, " grain");
```
3. Validating rice weight (rice weight must be between 10 and 100)

```c
do
    printf(" Input Weight of The Rice Sack [10..100 kg(s)]: ");
    scanf("%d", &stack[total].weight);
    fflush(stdin);
while(stack[total].weight<10 || stack[total].weight>100);
```

4. Increasing the number of data (total) since new data is put on the top of the stack.

```c
++total;
printf("\n\n--- Add Rice Sack Success ---\n");
```

9. Create Sell Method (pop)

Sell method will allow user to remove a data from the top of the stack and reveals previously concealed items, or results in an empty stack.

a. Checking whether the stack is empty

```c
void sell()
{
    if(total>U)
    {
        //removing a data from the top of the stack (2)
    }
    else
    {
        printf("\n\n--- The Rice Storage is Empty ---\n");
    }
getchar();
flush(stdin);
}
```

b. Removing a data from the top of the stack

```c
--total;
strcpy(stack[total].type, "");
stack[total].weight = 0;
printf("\n\n--- Sell Rice Sack Success ---\n");
```
10. Create Menu Method

Menu method will show menu and call view method.

```c
void menu()
{
    printf("\n BLUE RICE STOCK\n");
    printf("\n ^^^^^^^^^^^^^^^\n\n") ;
    view(); // calling view method
    printf("\n\n Option : ");
    printf("\n 1. Stock Rice Sack\n");
    printf("\n 2. Sell Rice Sack\n");
    printf("\n 3. Exit\n");
    printf("\n\n >> Input choice : ");
}
```

11. Create Main Method

Main method will allow user to input a choice what they want to do (stock, sell, or exit).

```c
int main()
{
    int choice;
    do
    {
        clear();
        menu();
        scanf("%d", &choice);
        fflush(stdin);

        switch(choice)
        {
            case 1 : stock(); // calling stock method
                     break;
            case 2 : sell();  // calling sell method
                     break;
        }
    }while(choice!=3);
    return 0;
}
```
2.4. Exercise

Mr. Ali Use is a seller of daily needs. Due to his rice stock system is not too good, he asks you to make a program using a stack with a concept of array of struct. Here are the descriptions of the program:

- Program always show rice stock in stack view.
- Program consists of 3 menus:
  1. Stock Rice Sack
  2. Sell Rice Sack
  3. Exit
- If user chooses **Stock Rice Sack**, then:
  - Ask user to input **type of rice**. Validate that the **type of the rice** must be ‘long’, ‘medium’, or ‘short’ with case sensitive. Then add the **type of the rice** with ‘grain’ automatically.
  - Ask user to input **weight of rice sack**. Validate that the weight must be between 10 and 100 kilograms.
  - Maximum data that can be stored in the rice stack is only 10. If the data has reached 10, then show the message “--- The Rice Storage is Full ---”
  - If data has been successfully inputted, show the message “--- Add Rice Sack Success ---”
- If user chooses **Sell Rice Sack**, then:
  - If there is no data in linked list, then show the message “--- The Rice Storage is Empty ---”.
  - If data is already in the linked list, the program will delete the last data in the stack and show the message “--- Sell Rice Sack Success ---”
- If user chooses **Exit**, then delete all data and the program will ends.

Print Screen of Main Menu

```
BLUE RICE STOCK
~~~~~~~~~~~~~~~~~~

Rice Stock <STACK>

Option :
1. Stock Rice Sack
2. Sell Rice Sack
3. Exit

>> Input choice :
```
Print Screen of Stock Rice Sack Menu (Menu ‘1’)

Input Rice Type [long/medium/short] grain: long
Input Weight of The Rice Sack [10..100 kg(s)]: 89

--- Add Rice Sack Success ---

Print Screen of Main Menu after Doing Stock Rice Sack Menu

BLUE RICE STOCK

Rice Stock (STACK)
[ long grain |  89 kg(s) ] -> [top]

Option:
1. Stock Rice Sack
2. Sell Rice Sack
3. Exit

>> Input choice:

Print Screen of Main Menu after Doing More Stock Rice Sack Menu

BLUE RICE STOCK

Rice Stock (STACK)
[ short grain |  65 kg(s) ] -> [top]
[ medium grain |  90 kg(s) ]
[ long grain |  89 kg(s) ]

Option:
1. Stock Rice Sack
2. Sell Rice Sack
3. Exit

>> Input choice:

Print Screen of Sell Rice Sack Menu (Menu ‘2’)

--- Sell Rice Sack Success ---
Print Screen of Main Menu after Doing Sell Rice Sack

BLUE RICE STOCK
~~~~~~~~~~~~~~~~~

Rice Stock <STACK>

[ medium grain ] 90 kg(s) ] -> [top]
[ long grain ] 89 kg(s) ]

Option:
1. Stock Rice Sack
2. Sell Rice Sack
3. Exit

>> Input choice:
Chapter 03
Queue
3.1. Definition of Queue

**Queue** is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a **FIFO** (First In First Out).

In a FIFO data structure, **the first element added to the queue will be the first one to be removed.** This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

3.2. Differences of Stack and Queue

- The big differences of **stack** and **queue** is that **stack** using **LIFO** (Last In First Out), while queue is using **FIFO** (First In First Out).
- Item can be added or removed only at one end (**top**) in **stack** and in a queue insertion at the **rear** and deletion from the **front**.
- The basic operation of stack are '**push**' and '**pop**', on other hand of queue are 'enqueue' and 'dequeue'.

Read more: [http://wiki.answers.com/Q/What_are_the_Differences_between_stack_and_queue#ixzz1JdwoQgRb](http://wiki.answers.com/Q/What_are_the_Differences_between_stack_and_queue#ixzz1JdwoQgRb)

Example:

You can think of a **queue** like a line at the bank. The first person in the queue will come to the teller first. If a bunch of people come while all the tellers are busy, they stand in line in the order in which they arrived. That is to say, new people (items) are added to the end of the line and the first person in line is the only one who is
called to a teller. In real life this is known as "first come, first served." In programming terms it's known as first-in-first-out, or **FIFO**.

You can think of a **stack** like a deck of cards. You can put down cards into a pile on your table one at a time, but if you want to draw cards, you can only draw them from the top of the deck one at a time. Unlike a queue, the first card to be put down is the last card to be used. This is known as first in last out, or **FILO** (also called **LIFO** for last in first out).

Read more:
http://wiki.answers.com/Q/What_is_the_difference_between_a_queue_and_a_stack
#ixzz1Jdx7lKlp

### 3.3. Enqueue and Dequeue Operation

#### 3.3.1. Enqueue

**Enqueue** is an operation in queue to add new data to the queue. In **stack** we called it “**Push**”. Push and Enqueue have same meaning but to show the differences between stack and queue, push in queue named as Enqueue.

#### 3.3.2. Dequeue

**Dequeue** is an operation in queue to delete data from the queue. In **stack** we called it “**Pop**”. Pop and Dequeue have same meaning but to show the differences between stack and queue, pop in queue named as Dequeue.
3.4. Example

3.4.1. Example 03 – Blue Tire Shop

Mr. Battlax is a manager of Blue Tire Shop. He wants to manage his tire stock better. So he asks you as a skilful programmer to make a simple circular queue program using array. Here are the descriptions of the program:

- Program consists of 4 menus:
  1. View Queue
  2. Queue Tire
  3. Get Tire from Queue
  4. Exit

- If user chooses View Queue, then:
  o If there is no data in queue, show the message “--- There is No Tire in The Queue ---”
  o If data is already in the queue, show the queue of tire.

- If user chooses Queue Tire, then:
  o If the number of tire is less than 10, ask user to input the pattern of tire. Validating the pattern of tire must be in non-alphanumeric character (*, ^, etc). However, if the number of tire is equals to 10, show the message “--- Queue is Full ---”
  o If the data has been inputted, then add the number of tire and show the message “--- Queue Tire Success ---”

- If user chooses Get Tire from Queue, then:
  o If there is no data in queue, show the message “--- There is No Tire in The Queue ---”
  o If data is already in the queue, the program will remove one data using the queue concept from the queue. If the data has been removed, show the message “--- Get Tire Success ---”

2. If user chooses Exit, then program ends.
3.4.2. Explanation

1. Create solution named “Soal03” and save to Drive D:\

2. Go To Soal03.cpp on Solution Explorer and double click on file

3. Type include stdio.h file and ctype.h file

```cpp
#include <stdio.h>
#include <ctype.h>
```

4. Declare size of queue

```cpp
#define SIZE 10
```

5. Declare struct data as tire and the array of struct. The struct has properties on following:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>char</td>
<td>Pattern of tire</td>
</tr>
</tbody>
</table>

```cpp
struct data
{
    char pattern;
} tire[10];
```

6. Declare global variable to point index of queue

```cpp
int head = 0, tail = 0, curr = 0;
```

7. Create Clear Method

Clear method will clear the screen with enter character (“\n”) 25 lines

```cpp
void clear()
{
    int i;
    for(i=0;i<25;i++) printf("\n");
}
```
8. Create Push Method

Push method will allow user to push the pattern of the tire into the queue. If the queue is empty, first tire (head) will be the same as the last tire (tail), but if the queue is not empty, the inputted tire will be the last tire (tail).

```java
void push(char pattern)
{
    if(tire[head].pattern == NULL)
    {
        tire[head].pattern = tire[tail].pattern = pattern;
    }
    else
    {
        tail = ++tail % SIZE;
        tire[tail].pattern = pattern;
    }
}
```

9. Create Pop Method

Pop method will allow user to pop the first tire (head) from queue. If there is only one tire in the queue tail will be the same as head, but if there are more than one tire in the queue, the first tire (head) will be moved and changed to the next tire in the queue.

```java
void pop()
{
    tire[head].pattern = NULL;
    if(head == tail)
    {
        tail = head = ++head % SIZE;
    }
    else
    {
        head = ++head % SIZE;
    }
}
```

10. Create View Method

View method will allow user to view the queue of tire. This method will clear the screen first and do some checking before viewing the queue of tire.

```java
void view()
{
    clear();
}
```
a. Check on the queue condition, if it is empty then system will print "\n\n--- There is No Tire in The Queue ---\n"

```
if(tire[head].pattern == NULL)
{
    printf("\n\n--- There is No Tire in The Queue ---\n");
}
```

b. Otherwise, system will print "\n - Queue of Tire -\n\n" and view all queue of tires

```
else
{
    printf("\n - Queue of Tire -\n\n");
curr = head-1;
do
{
    curr = ++curr % SIZE;
    printf("[\n");
    for(int i=0;i<10;i++)
    {
        printf("%c", tire[curr].pattern);
    }
    printf("]\n");
    printf("\n");
}while(corr != tail);
```

c. Hold screen until user hits enter button

```
getchar();
fflush(stdin);
```

11. Create Queue Tire Method

Queue tire method will allow user to add (push) a new tire into the queue. This method will check the size of the queue first before queueing a new tire.

```
void queueTire()
{
    char pattern;
    printf("\n\n\n");
}
```

a. Check on the condition of the queue, if it is full then system will print "\n\n --- Queue is Full ---\n"

```
if((tail+1) % SIZE == head)
{
    printf("\n\n --- Queue is Full ---\n");
}
```
b. Otherwise, system will ask user to input one non-alphanumerical character as the pattern of tire, push it into the queue, then print

“\n\n\n --- Queue Tire Success ---\n”

```c
else
{
    do
    {
        printf(" Input The Pattern of Tire [nonalphanumeric char]: ");
        scanf("%c", &pattern);
        fflush(stdin);
    }while(isalpha(pattern) || isdigit(pattern) || pattern=='\n');
    push(pattern);
    printf("\n\n\n --- Queue Tire Success ---\n");
}
```

c. Hold screen until user hits enter button

```c
getchar();
fflush(stdin);
```

12. Create Get Tire Method

Get tire method will allow user to get tire from the queue and pop the first tire (head) from the queue.

This method will check the condition of the queue first before executing pop. If it is empty then system will print “\n\n\n --- There is No Tire in The Queue ---\n”. Otherwise, system will do the pop and print “\n\n\n --- Get Tire Success ---\n”

```c
void gettire()
{
    if(tire[head].pattern == NULL)
    {
        printf("\n\n\n --- There is No Tire in The Queue ---\n");
    }
    else
    {
        pop();
        printf("\n\n\n --- Get Tire Success ---\n");
    }
    getchar();
    fflush(stdin);
}
```
13. Create Menu Method

Menu method will allow user to print the available menus

```c
void menu()
{
    printf("\n BLUE TIRE SHOP\n");
    printf("\n xxxxxxxxxxxxxxx\n");
    printf("\n 1. View Queue\n");
    printf("\n 2. Queue Tire\n");
    printf("\n 3. Get Tire from Queue\n");
    printf("\n 4. Exit\n");
    printf("\n\n >> Input choice : ");
}
```

14. Create Main Method

The main method of the program

```c
int main()
{
    return 0;
}
```

a. Declare one variable choice to collect user selected menu

```c
int choice;
```

b. Create a main process loop in system. In this part, we will create a main loop process which will loop until user gives value 4 into choice variable

```c
while(choice!=4);
```

c. Clear the screen, view the menus then ask user to choose it

```c
clear();
menu();
scanf("%d", &choice);
fflush(stdin);
```
d. Create an option menu from user’s choice with switch case syntax with condition:
- If user choose first menu with value 1, then program will run view method and view all queue tires
- If user choose first menu with value 2, then program will run queue tire method to ask user to input the pattern of the tire
- If user choose first menu with value 3, then program will run get tire method to pop the first tire (head) in the queue

```java
switch(choice)
{
  case 1:
    view();
    break;
  case 2:
    queueTire();
    break;
  case 3:
    getTire();
    break;
}
```

3.5. Exercise

Mr. Brea Stanley is a famous dress designer. He wants to run a dress shop independently and he needs a program to support the cashier. Therefore, he asks you as a programmer to make a program using the queue concept. Here are the descriptions of the program:

- Program consists of 4 menus:
  1. View Queue
  2. Add Customer to Queue
  3. Serve Customer
  4. Exit

- If user chooses View Queue, show the queue view that contains No, Customer’s Name, Dress Name, and Price.

- If user chooses Add Customer to Queue, then:
  o Ask user to input customer’s name. It must be in alphabets only and the length must be between 3 and 20 characters.
Ask user to input dress name. The length must be between 3 and 20 characters.

Ask user to input dress price. The dress price must be between $50 and $999.

Maximum customer in the queue is 10. If it has reached 10 customers, then show the message “--- Maximum Customer in a Queue is 10 ---”

If not, then show the message “--- Success to Ad Customer into Queue List ---”

- If user chooses Serve Customer, then:
  - Program will delete the data using the queue concept.
  - If there is no data in the list, then show the message “--- There is No Customer in Queue ---”.
  - If data has been successfully deleted, then show the message “--- [customer’s name] Has Been Served ---”

- If user chooses Exit, then delete all data in the linked list and program will ends.

Print Screen of Main Menu

```
BLUE DRESS SHOP CASHIER QUEUE
=================================
1. View Queue
2. Add Customer to Queue
3. Serve Customer
4. Exit

>> Input choice :
```

Print Screen of View Queue Menu (Menu ‘1’)

```
--- QUEUE VIEW ---

<table>
<thead>
<tr>
<th>No.</th>
<th>Customer’s Name</th>
<th>Dress Name</th>
<th>Price</th>
</tr>
</thead>
</table>

---
```

Print Screen of Add Customer to Queue Menu (Menu ‘2’)

```
Input Customer’s Name [3..20][must be in alphabets]: Valiant Wyvern
Input Dress Name [3..20]: Top Blouse
Input Dress Price [$50..$999]: $67

--- Success to Add Customer into Queue List ---
```
Print Screen of View Queue Menu after Add Customer

--- QUEUE VIEW ---

<table>
<thead>
<tr>
<th>No.</th>
<th>Customer’s Name</th>
<th>Dress Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Valiant Wyvern</td>
<td>Top Blouse</td>
<td>$ 67</td>
</tr>
<tr>
<td>2</td>
<td>Pulpy</td>
<td>White Jeans UK</td>
<td>$ 50</td>
</tr>
<tr>
<td>3</td>
<td>Nice Chou</td>
<td>Extreme Shirt</td>
<td>$ 175</td>
</tr>
</tbody>
</table>

Print Screen of Serve Customer Menu (Menu ‘3’)

--- Valiant Wyvern Has Been Served ---

Print Screen of View Queue Menu (Menu ‘1’) After Choosing Serve Customer Menu

--- QUEUE VIEW ---

<table>
<thead>
<tr>
<th>No.</th>
<th>Customer’s Name</th>
<th>Dress Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pulpy</td>
<td>White Jeans UK</td>
<td>$ 50</td>
</tr>
<tr>
<td>2</td>
<td>Nice Chou</td>
<td>Extreme Shirt</td>
<td>$ 175</td>
</tr>
</tbody>
</table>
Chapter 04

Single Linked List
4.1. Definition of Single Linked List

**Single (Singly) Linked List** is a data structure that consists of sequence of nodes which contains a reference (link) to the next node in the sequence.

The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure.

Linked lists allow insertion and deletion of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link that being added or removed is maintained during the search list.

4.2. Example

4.2.1. Example 04 – Blue Computer Administrator

Mr. Jagi is a manager from a computer shop. One day Mr. Jagi wants to change the old stock administration system of his computer shop into a new system. So he asks you as a skilful programmer to make a program using a linked list concept. Here are the descriptions of the prototype:

- The program consists of 4 menus:
  1. Item List
  2. Add (PUSH) New Item
  3. Delete (POP) Item
  4. Exit
- If user chooses Item List, then program will show the item list that contains No, Name, Type, Quantity, and Price of the items.
- If user chooses Add (PUSH) New Item (menu 1), then:
  - Ask user to input the name of the new item that he/she wants to add to the list. The length of the new item name must be between 3 and 20 characters.
  - Ask user to input the type of the new item. The type of the new item must be between “processor”, “graphic card”, or “memory”.
  - Ask user to input quantity of the new item. The quantity must be between 1 and 20.
✓ Ask user to input the price of the new item. The price must be between $1 and $1000.
✓ If data has been successfully added, show the message “--- Add New Item Success ---”
- If user chooses Delete (POP) Item (menu 2), then ask user to input the name of the new item that he/she wants to delete from the list. The length of new item name must be between 3 and 20 characters.
- If user chooses Exit (menu 3), then delete all data in the linked list and the program will ends.

4.2.2. Explanation
1. Create solution named “Soal04” and save to drive D:\
2. Go To main.cpp on Solution Explorer and double click on file
3. Type include stdio.h, string.h, and stdlib.h file

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

2. Declare struct data as digipet data which is have properties on following:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Char 51</td>
<td>Name of item</td>
</tr>
<tr>
<td>Type</td>
<td>Char 21</td>
<td>Type of item</td>
</tr>
<tr>
<td>Qty</td>
<td>Integer</td>
<td>Number of item</td>
</tr>
<tr>
<td>Price</td>
<td>Integer</td>
<td>Price of item</td>
</tr>
<tr>
<td>Next</td>
<td>Struct data *</td>
<td></td>
</tr>
</tbody>
</table>

```
struct data
{
    char name[51];
    char type[21];
    int qty;
    int price;
    struct data *next;
};
```
3. Declare 3 pointer variable from struct data, such as: head, tail, and curr. A linked list is a chain of structs or records called nodes.

```c
struct data
{
    char name[51];
    char type[21];
    int qty;
    int price;
    struct data *next;
} *head, *tail, *curr;
```

4. Create View Method

View method will allow user to view the item list that contains No, Name, Type, Quantity, and Price of the items.

```c
void view()
{
    int i = 0, j = 0;
    // clear();
    printf("--- ITEM LIST ---\n\n");
    for(j = 0; j < 68; j++)
        printf("=\n");
    printf("\n\n");
    printf("| %-3s | %-20s | %-15s | %-8s | %-4s |\n",
        "No.", "Name", "Type", "Quantity", "Price");
    for(j = 0; j < 68; j++)
        printf("=\n");
    printf("\n\n");
    curr = head;
    while(curr)
    {
        i++;
        printf("| %d | %-20s | %-15s | %d | %d |\n",
            i, curr->name, curr->type, curr->qty, curr->price);
        curr = curr->next;
    }
    for(j = 0; j < 68; j++)
        printf("=\n");
    printf("\n");
    getchar();
    fflush(stdin);
}
5. Create Push Method

Push Method is used for insert a new data into a linked list. Insertion into a single-linked list has two special cases: insertion a new node before the head (to the very beginning of the list) and insertion after the tail (to the very end of the list).

In this case, we will learn about insertion a new node after the tail.

```c
void push(char name[], char type[], int qty, int price)
{
    curr = (struct data*)malloc(sizeof(struct data));
    strcpy(curr->name, name);
    strcpy(curr->type, type);
    curr->qty = qty;
    curr->price = price;
    if(head == NULL)
    {
        head = tail = curr;
    }
    else
    {
        tail->next = curr;
        tail = curr;
    }
    tail->next = NULL;
}
```

6. This method takes four parameters and passing by value.

```c
void push(char name[], char type[], int qty, int price)
```

7. Allocation memory for a new node.

```c
curr = (struct data*)malloc(sizeof(struct data));
```

8. Insertion data from parameter into a new node.

```c
strcpy(curr->name, name);
strcpy(curr->type, type);
curr->qty = qty;
curr->price = price;
```
9. Check condition of the linked list:
   a. If list is empty, which is indicated by `(head == NULL)` condition. Set both head and tail to point to the new node.

   ```
   if (head == NULL) {
       head = tail = curr;
   }
   ```

   ![Diagram showing head and tail being set to the new node](image)

   b. If list is not empty, new node is inserted left after the current tail node.

   ```
   else {
       tail->next = curr;  \(1\)
       tail = curr;        \(2\)
   }
   ```

   ![Diagram showing new node inserted after the current tail node](image)

   c. Set the pointer of tail to NULL

   ```
   tail->next = NULL;
   ```

   ![Diagram showing tail being set to NULL](image)
10. Create Pop Method

Pop method will allow user to delete item name which is inserted.
This method take one parameter array of char and passing by value.

```c
void pop(char name[])
{
    curr = head;
    while (curr != NULL && strcmp(curr->name, name) != 0)
    {
        curr = curr->next;
    }
    if (curr == NULL)
    {
        printf("Item name that you inserted cannot be found.\n");
    }
    else if (head == curr)
    {
        head = head->next;
        free(curr);
        printf("Item deleted successfully.\n");
    }
    else if (head == tail)
    {
        printf("The last Item cannot be Deleted\n");
    }
    else
    {
        struct data * temp;
        temp = head;
        while (temp->next != curr) temp = temp->next;
        temp->next = curr->next;
        free(curr);
        printf("Item deleted successfully.\n");
    }
}
```

11. Set curr node to another node in the list that user want to delete

```c
curr = head;
while (curr != NULL && strcmp(curr->name, name) != 0)
{
    curr = curr->next;
}
```

Example:
If we want to delete item name "xxx"

```
Data1 → Data2 → xxx → Data4 → NULL

head ← curr
```

```
curr ← curr
```
curr node will looping from head node until the node that contain name of item "xxx"

12. Check position of curr:
   a. If item name is not exist in the list, it is indicated by \( (curr == NULL) \) condition.
      Example:
      If we want to delete item name "xxx" and item name does not exists in linked list
      ```
      else if(head==curr)
      {  
          head = head->next;  
          free(curr);       
          printf(" Item deleted successfully\n");  
      }
      ```

   b. If item name that user wants to delete is the first node in the list, it is indicated by \( (head == curr) \) condition.
      Example:
      If we want to delete item name "xxx" and item name is the first node
c. If the node position of the item name that user wants to delete is between head and tail, it is indicated by (head!=curr && tail!=curr) condition

```c
else if(head!=curr && tail!=curr)
{
    struct data * temp;
    temp=head;
    while(temp->next!=curr) temp=temp->next; 1
    temp->next = curr->next; 2
    free(curr);
    printf(" Item deleted successfully.");
}
```

Example:
If we want to delete item name “xxx” and position of item name is between head and tail.
Set new variable called “temp” before curr node.

---

d. If item name that user wants to delete is the last node in list, it is indicated by (tail == curr) condition.
13. Create Pop All Method

Pop All method will delete all nodes in the list

```c
void popall()
{
    while(head!=NULL)
    {
        curr=head;
        head=head->next;
        free(curr);
    }
}
```

14. Create Menu Method

```c
void menu()
{
    printf("\n BLUE COMPUTER ADMINISTRATOR\n");
    printf("\n +++++++++++++++++++++++++++++++\n");
    printf("\n 1. Item List\n");
    printf("\n 2. Add (PUSH) New Item\n");
    printf("\n 3. Delete (POP) Item\n");
    printf("\n 4. Exit\n");
    printf("\n >> Input your choice : ");
}
```

15. Create Clear Method

Clear method will clear the screen with enter character ("\n") 25 lines

```c
void clear()
{
    for(int i=0;i<25;i++)
        printf("\n");
}
```

16. Create Main Method

a. Declare five variables to collect user input.

```c
int choice;
char name[51];
char type[21];
int qty;
int price;
```
b. Create a main process loop in system. In this part we will create a main loop process which will loop until user gives value 4 into variable “choice”.

```c
    do
    {
        while(choice!=4);
    }
```

c. Create a simple menu

```c
do
{
    clear();
    menu();
    scanf("%d", &choice);
    fflush(stdin);
    switch(choice)
    {
        case 1 : break;
        case 2 : break;
        case 3 : break;
    }
} while(choice!=4);
```

d. Create menu **Item List** (menu ‘1’)

```c
    case 1 : view();
    break;
```

e. Create menu **Add (PUSH) New Item** (menu ‘2’)

```c
    case 1 :
    do
    {
        printf(" Input Name Of The New Item [1..20]: ");
        scanf("%[^\n]", name);
        fflush(stdin);
        while(strlen(name)<3 || strlen(name)>20);
        printf("\n");
        do
        {
            printf(" Input Type Of The New Item [processor/graphic card/memory]: ");
            scanf("%[^\n]", type);
            fflush(stdin);
            while(strcmp(type,"processor")!=0 && strcmp(type,"graphic card")!=0 && strcmp(type,"memory")!=0);
            printf("\n");
```
f. Create menu **Delete (POP) Item** (menu ‘3’)

```c
case 3:
    if (head==NULL)
        
    else
        do
            printf(" Input Name of The New Item [3..20]: ");
            scanf("%[^\n]", name);
            fflush(stdin);
        )while(strlen(name)<3 || strlen(name)>20);
        pop(name);
    )
    getchar();
    fflush(stdin);
    break;
```

g. If user chooses menu **Exit** (menu ‘4’), then call the Pop All Method.

```c
popall();
```
Chapter 05
Double Linked List
5.1. Definition of Double Linked List

Double Linked List is a linked data structure that consists of a set of data records/nodes, each having two special link fields that contain references to the previous and to the next record in the sequence.

It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders.

The two links allow walking along the list in either direction with equal ease.

Compared to a single linked list, modifying a double linked list usually requires changing more pointers, but is sometimes simpler because there is no need to keep track of the address of the previous node.

5.2. Example

5.2.1. Example 01 – Blue Motorcycle Parts

Ms. Aikawa wants to run a new business, “motorcycle parts”. He wants to make a program to support his business operation. So he asks you as his good friend to make a program using the double linked list concept. Here are the descriptions of the program:

- Program consists of 4 menus:
  - View Order List
  - Add A New Order
  - Take Order
  - Exit
- If user chooses View Order List (menu 1), then show the list of ordered motorcycle parts.
- If user chooses Add New Order (menu 2), then:
- Ask user to input name of motorcycle’s part that he/she wants to purchase. Validate the length **must be between 3 and 30 characters**.
- Ask user to input quantity of motorcycle’s parts that he/she wants to purchase. Validate the quantity **must be between 1 and 20**.
- If the data has been successfully inputted, then show the message “--- Add New Order Success ---”

- If user chooses **Take Order**, then:
  - If there is no data in the linked list, then show the message “--- There is No Order in The List ---”
  - If data is already in the linked list, the program will show the list of ordered motorcycle parts and ask the user to input the number of the order that he/she wants to take. Validate that user **must input between 1 and the total number of all parts in the list**.
  - If the data has been successfully taken, then show the message “--- Take Order Success ---”

- If user chooses **Exit**, then delete all data in the linked list and the program will ends.

5.2.2. Explanation

1. Create solution “Answer05” and save to drive D:
2. Go to Answer05.cpp on Solution Explorer and double click on file

![Solution 'Answer05' (1 project)]
- [Answer05]
- [Header Files]
- [Resource Files]
- [Source Files]
- [Answer05.cpp]

3. Type include stdio.h, string.h, and stdlib.h

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```
4. Declare array of struct to store motorcycle part and its quantity.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>part</td>
<td>char[50]</td>
<td>Motorcycle part</td>
</tr>
<tr>
<td>qty</td>
<td>int</td>
<td>Motorcycle part quantity</td>
</tr>
<tr>
<td>prev</td>
<td>struct data *</td>
<td>Pointer referencing to previous data</td>
</tr>
<tr>
<td>next</td>
<td>struct data *</td>
<td>Pointer referencing to next data</td>
</tr>
</tbody>
</table>

5. Declare 3 global pointers: **head**, **tail**, and **curr**.

**Head** pointer always references to the **first** node;

**Tail** pointer always references to the **last** node.

```c
struct data
{
    char part[50];
    int qty;
    struct data *next, *prev;
}*head, *tail, *curr;
```

6. Create View Method

View method will show all items/data in the linked list. It will show all data from the first node to the last one.

This calls a clear method to print 25 lines and clear the screen.

```c
void clear()
{
    int i;
    for(i=0;i<25;i++) printf("\n");
}

void view()
{
    int i = 0;
    clear();
    printf(" ORDER LIST ---\n");
    printf(" +-------------------+\n");
    printf(" | No. | Name of Parts | Quantity |\n");
    printf(" +-------------------+\n");
    curr = head; //moving pointer to first node
    while(curr)
    {
        i++;
        printf(" | %3d. | %-30s | %8d |\n", i, curr->part, curr->qty);
        curr = curr->next; //moving pointer from current node to next node
    }
    printf(" +-------------------+\n");
}"
7. Create Push Head Method

Push head method will allow user to add a new node to the linked list. The added node will be inserted before the first node (head).

```c
void pushhead(char part[], int qty)
{
    curr = (struct data*)malloc(sizeof(struct data));
    strcpy(curr->part, part);
    curr->qty = qty;

    if(head==NULL)
    {
        head = tail = curr;
    }
    else
    {
        curr->next = head;
        head->prev = curr;
        head = curr;
    }
    head->prev = NULL;
    tail->next = NULL;
}
```

8. The method has 2 parameters to fill

```c
void pushhead(char part[], int qty)
```

9. Allocating memory for a new node

Sizeof (struct data) will return struct size so the memory allocated will be fit exactly with the memory needed for a struct.

```c
curr = (struct data*)malloc(sizeof(struct data));
```

10. Inserting data from parameters into a new node

Strcpy method needs an additional library: string.h

```c
strcpy(curr->part, part);
curr->qty = qty;
```
11. Checking the condition of the linked list
   a. If the list is still empty, which is indicated by the position of head (refering to nowhere memory/NULL), set both head and tail pointer to reference to the new node.

```c
if (head == NULL)
{
    head = tail = curr;
}
```

b. Otherwise, insert new node before the first data (Remember: head must always references to the first node and tail must always references to the last node)

```c
else
{
    curr->next = head;  1
    head->prev = curr;  2
    head = curr;  3
}
```

12. Pointing unused pointer to NULL

```c
head->prev = NULL;
tail->next = NULL;  4
```
13. Create Pop Method

Pop method will allow user to remove a node from the list. This method has 4 possible conditions of the list. Each condition has its own way to remove the node.

a. Checking if the list is empty

```c
void pop()
{
    struct data *temp;
    if(head!=NULL)
    {
        // point 2
    }
}
```

b. Removing a node from the list

- Removed node is the first node (pop head)

```c
if(curr == head)
{
    head = head->next;
    free(curr);
    //to prevent error (in case the list has only 1 node)
    if(head != NULL)
    {
        head->prev = NULL;
    }
}
```
- Removed node is the last node (pop tail)

```c
else if (curr == tail) {
    curr = tail;
    tail = tail->prev;
    free(curr);
    tail->next = NULL;
}
```
- Removed node is on the middle (pop middle)

```c
else
{
    temp = head;
    while(temp->next != curr)
    {
        temp = temp->next;
    }
    curr->next->prev = curr->prev;
    temp->next = curr->next;
    free(curr);
}
```

15. Create Popall Method

Popall method will delete all nodes in the list

```c
void popall()
{
    while(head!=NULL)
    {
        curr=head;
        head=head->next;
        free(curr);
    }
}
```
16. Create AddOrder Method

Addorder Method will allow user to input motorcycle part and quantity and then insert it into the list by calling pushhead method.

```c
void addorder()
{
    char part[51];
    int qty;
    printf("\n\n\n");
    do
    {
        printf(" Input Name of Motorcycle’s Part [3..30]: ");
        scanf("%[^\n]", part);
        fflush(stdin);
    } while(strlen(part)<3 || strlen(part)>30);
    printf("\n");
    do
    {
        printf(" Input Quantity of The Motorcycle’s Part [1..20]: ");
        scanf("%d", &qty);
        fflush(stdin);
    } while(qty<1 || qty>20);
    pushhead(part, qty);
    printf("\n\n\n --- Add New Order Success ---\n");
    getchar();
}
```

17. Create Delete Order Method

Delete order method will allow user to choose which data to delete by inputting a data number.

a. Checking if the list is empty

```c
void deleteorder()
{
    int tot = 0;
    int pos;
    int i;
    if(head==NULL)
    {
        //if list is empty
        printf("\n\n\n --- There is No Order in The List ---\n");
    }
    else
    {
        //counting the number of nodes in the list (2)
        //validating the number of the order (3)
        //moving curr pointer to node, which will be deleted (4)
        pop(); //calling pop method
        printf("\n\n\n --- Take Order Success ---\n");
    }
    getchar();
    fflush(stdin);
}
```
b. Counting the number of nodes in the list

```c
view();
curr = head;
while (curr)
{
    tot++;
    curr = curr->next;
}
```

c. Validating Number of The Order

```c
printf("\n\n");
do
{
    printf(" Input Number of The Order [1..%d]: ", tot);
    scanf("%d", &pos);
    fflush(stdin);
} while (pos<1 || pos>tot);
```

d. Moving curr pointer to node which will be deleted

```c
curr = head;
for (i=1; i<pos; i++)
{
    curr = curr->next;
}
```

18. Creating Menu Method

```c
void menu()
{
    printf("\n BLUE MOTORCYCLE PARTS\n");
    printf("\n  . . . . . . . . . . . \n");
    printf("\n  1. View Order List\n");
    printf("\n  2. Add New Order\n");
    printf("\n  3. Take Order\n");
    printf("\n  4. Exit\n");
    printf("\n\n>> Input choice : ");
}
```
19. Main Method

```c
int main()
{
    int choice;
    do
    {
        clear();
        menu();
        scanf("%d", &choice);
        fflush(stdin);
        switch(choice)
        {
            case 1: view();
                   getchar();
                   fflush(stdin);
                   break;
            case 2: addorder();
                   break;
            case 3: deleteorder();
                   break;
        }
    }while(choice!=4);
    popall();
    return 0;
}
```
Chapter 06
Binary Tree
6.1. Definition of Binary Tree

Binary tree is a tree data structure in which each node had at most two child nodes, usually distinguished as "left" and "right". Nodes with children are parent nodes, and child nodes may contain references to their parents.

Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists. Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right child.

Binary trees are used to implement binary search trees and binary heaps.

6.2. Operation in Binary Tree

6.2.1. Insertion

Nodes can be inserted into binary trees in between two other nodes or added after an external node/leaf node (node of a tree data structure that has zero child nodes).

In binary trees, a node that is inserted is specified as to which child it is.

1. Insert to be child on external node

   If the tree have an external node called “A”, then to add new node called “B” after node “A”.

   “A” will assigns the new node as one of its children and the new node assigns node “A” as its parents.

2. Insert to be child on internal node

   Insertion on internal nodes is slightly more complex than on external nodes. Say that the internal node is node “A” and that node “B” is the child of “A”. (If the insertion is to insert a right child, then “B” is the right child of “A”, and similarly with a left child insertion.)

   “A” assigns its child to the new node and the new node assigns its parent to “A”.

   ![Binary Tree Insertion Diagram]
Then the new node assigns its child to "B" and "B" assigns its parent as the new node.

6.2.2. Deletion

Deletion is the process whereby a node is removed from the tree. Only certain nodes in a binary tree can be removed unambiguously.

1. Node with zero or one children

Say that the node to delete is node A.

If a node has no children (external node), deletion is accomplished by setting the child of "A"'s parent to null and "A"'s parent to null.

If it has one child, set the parent of "A"'s child to "A"'s parent and set the child of "A"'s parent to "A"'s child.

2. Node with two children

In a binary tree, a node with two children cannot be deleted unambiguously. However, in certain binary trees these nodes can be deleted, including binary search trees.

6.2.3. Iteration

Often, one wishes to visit each of the nodes in a tree and examine the value there, a process called iteration or enumeration.

There are several common orders in which the nodes can be visited, and each has useful properties that are exploited in algorithms based on binary trees:
• Pre-Order: Root first, children after
  To traverse a non-empty binary tree in **preorder**, perform the following operations recursively at each node, starting with the root node:
  ✓ Visit the root.
  ✓ Traverse the left subtree.
  ✓ Traverse the right subtree.

• Post-Order: Children first, root after
  To traverse a non-empty binary tree in **inorder** (**symmetric**), perform the following operations recursively at each node:
  ✓ Traverse the left subtree.
  ✓ Visit the root.
  ✓ Traverse the right subtree.

• In-Order: Left child, root, right child.
  To traverse a non-empty binary tree in **postorder**, perform the following operations recursively at each node:
  ✓ Traverse the left subtree.
  ✓ Traverse the right subtree.
  ✓ Visit the root.

6.2.4. **Depth-first Order**

In depth-first order, we always attempt to visit the node farthest from the root that we can, but with the caveat that it must be a child of a node we have already visited. Unlike a depth-first search on graphs, there is no need to remember all the nodes we have visited, because a tree cannot contain cycles. Pre-order is a special case of this.

Result: A – B – D – E – C – F – G
6.2.5. Breadth-first Order

Contrasting with depth-first order is breadth-first order, which always attempts to visit the node closest to the root that it has not already visited. Also called a level-order traversal.

Result: A – B – C – D – E – F – G
6.3. Example

6.3.1. Example 06 – Blue Football Club

Mr. Joe is a manager of a football club named Blue. Blue is a developing football club, so it has a lot of football player coming in and out. He wants to make the task of managing the player transfer not too complicated. He asks you as a skilful programmer to make a program using the binary tree concept. Here are the descriptions of the program:

* Program consists of 3 menus:
  1. View All Player
  2. Add Player
  3. Exit and Pop All

* If user chooses View All Player (Menu 1), then:
  1. If there is no data in the tree, show the message “--- There is No Player in The Tree ---”
  2. If data is already in the tree, show the player list in this format:
    “Player List:
    [Player’s Name] ([Player’s Back Number])”

* If user chooses Add Player (Menu 2), then:
  1. Ask user to input player’s name. Validate that the length of player’s name must be between 3 and 20 characters.
  2. Ask user to input player’s back number. Validate that the player’s back number must be between 1 and 99.
  3. If the player’s back number already exists, show the message “* Player’s Back Number Cannot be the Same, Please Input Other Number *”
  4. If tree is still empty, then data will be inserted automatically.
  5. Otherwise, ask the user to input the direction where the data will be placed. Validate that the direction must be between “left” and “right”.
    1. If the direction chosen is “left”, the data will be pushed to the left of current node.
    2. If the direction chosen is “right”, the data will be pushed to the right of current node.
✓ Maximum tree level is 4. If level is already at maximum, show the message “--- Maximum Tree Level is 4 ---”
✓ If data has been successfully inputted, show the message “--- Add Player Success ---”
• If user chooses Exit and Remove All, then delete all data in the linked list and the program will ends.

6.3.2. Explanation
1. Create solution named “Soal06” and save to Drive D:\
2. Go To main.cpp on Solution Explorer and double click on file
3. Type include stdio.h, string.h and stdlib.h file

```cpp
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

3. Declare struct data as player data which is have properties on following:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Char [51]</td>
<td>Player’s name</td>
</tr>
<tr>
<td>number</td>
<td>Integer</td>
<td>Player’s number</td>
</tr>
<tr>
<td>left</td>
<td>Struct data *</td>
<td>Left subtree</td>
</tr>
<tr>
<td>right</td>
<td>Struct data *</td>
<td>Right subtree</td>
</tr>
</tbody>
</table>

```cpp
struct data
{
    char name[51];
    int number;
    struct data *left, *right;
}*root;
```
4. Declare 1 global variable
   This variable is used as a flag whether the data that will be inserted is already exist is the tree or not.

   ```cpp
   bool exist = false;
   ```

5. Create Clear Method
   Clear method is used to clear the screen by printing new line 25 times.

   ```cpp
   void clear()
   {
       int i;
       for(i=0;i<25;i++) printf("\n");
   }
   ```

6. Create Push Method
   Push method is used to insert a new data into the tree.

   ```cpp
   void push(struct data **temp, char name[], int number, int level)
   {
       char direction[100];
       if(level<4)
       {
           if(*temp==NULL)
           {
               *temp = (struct data *)malloc(sizeof(struct data));
               strcpy(*temp)->name, name);
               (*temp)->number = number;
               (*temp)->left = (*temp)->right=NULL;
               printf("\n\n\n --- Add Player Success ---\n");
           }
           else
           {
               printf("\n");
               do
               {
                   printf(" Will He Be in 'left' or 'right' %s ? ": (*temp)->name);
                   scanf("%s", direction);
                   fflush(stdin);
               }while(strcmp(direction,"left")!=0 &\& strcmp(direction,"right")!=0);
               if(strcmp(direction,"right")==0)
               {
                   push(*temp)->right, name, number,level+1);
               }
               else
               {
                   push(*temp)->left, name, number,level+1);
               }
           }
       }
   ```
Example of the tree:

a. This method takes four parameters. `temp` is passing by reference and the others are passing by value.

```c
void push(struct data **temp, char name[], int number, int level)
```

b. Declare a direction variable that is used to store the direction of putting data, left or right.

```c
char direction[100];
```

c. Variable `level` is used to count the number of levels in a tree. In this case, the tree is only consists of 4 levels.

```c
if(level<4)
{
}
else
{
    printf("\n\n\n--- Maximum Tree Level is 4 ---\n");
}
```
7. Check conditions of the node:
   a. If node is empty, which is indicated by (temp == NULL) condition. Insert data name and number from parameters into the tree.
   
   ```c
   if(*temp==NULL)
   {
       *temp = (struct data *)malloc(sizeof(struct data));
       strcpy((*temp)->name, name);
       (*temp)->number = number;
       (*temp)->left = (*temp)->right=NULL;
       printf("\n\n\n --- Add Player Success ---\n");
   }
   
   if(*temp==NULL)
   {
       *temp = (struct data *)malloc(sizeof(struct data));
       strcpy((*temp)->name, name);
       (*temp)->number = number;
       (*temp)->left = (*temp)->right=NULL;
       printf("\n\n\n --- Add Player Success ---\n");
   }
   
   if(*temp==NULL)
   {
       *temp = (struct data *)malloc(sizeof(struct data));
       strcpy((*temp)->name, name);
       (*temp)->number = number;
       (*temp)->left = (*temp)->right=NULL;
       printf("\n\n\n --- Add Player Success ---\n");
   }
   
   if(*temp==NULL)
   {
       *temp = (struct data *)malloc(sizeof(struct data));
       strcpy((*temp)->name, name);
       (*temp)->number = number;
       (*temp)->left = (*temp)->right=NULL;
       printf("\n\n\n --- Add Player Success ---\n");
   }
   ```
   
   b. If node is not empty, ask user which direction the data should be putted, left or right.
   
   ```c
   do
   {
       printf(" Will He Be in 'left' or 'right' %s ?: ", (*temp)->name);
       scanf("%[^\n]", direction);
       fflush(stdin);
   }while(strcmp(direction,"left")!=0 && strcmp(direction,"right")!=0);
   ```
   
   c. If user chooses “right”, push the data into the right subtree and increase the level by 1.
   
   ```c
   if(strcmp(direction,"right")==0)
   {
       push(&(*temp)->right, name, number, level+1);
   }
   ```
   
   d. Otherwise (user chooses left), push the data into the left subtree and increase the level by 1.
   
   ```c
   else
   {
       push(&(*temp)->left, name, number, level+1);
   }
   ```
8. Create Search Method

Search method will be used to check whether the data (player’s number) that will be inserted is already exist in the tree or not.

```c
void search(struct data **temp, int number)
{
    if((*temp)!=NULL && number==(*temp)->number)
    {
        exist = true;
    }
    if((*temp)!=NULL)
    {
        search(&(*temp)->left, number);
        search(&(*temp)->right, number);
    }
}
```

9. Create Popall Method

Popall method will be used to empty the tree and return all resources that is used to memory.

```c
void popall(struct data *temp)
{
    if(temp != NULL)
    {
        popall(temp->left);
        popall(temp->right);
        free(temp);
    }
}
```

10. Create Print Player Method

Print player method will be used to print all player’s data that is exist in the tree.

```c
void printplayer(struct data **temp)
{
    if((*temp)!-NULL)
    {
        printf(" - %20s (%d)\n", (*temp)->name, (*temp)->number);
        printplayer(&(*temp)->left);
        printplayer(&(*temp)->right);
    }
}
```
11. Create View Method

View Method will be used to view all player in the tree.

If the tree is not empty, which is indicated by (root != null) conditions, this method will call printplayer method to print all player’s data.

```c
void view()
{
    clear();
    if(root != NULL)
    {
        printf("\n Player List :\n");
        printplayer(&root);
    }
    else
    {
        printf("\n\n --- There is No Player In The Tree ---\n");
    }
    getchar();
    fflush(stdin);
}
```

12. Create Add Player Method

Add player method will be used to add a new player’s data into the tree if the tree is not full.

```c
void addplayer()
{
    char name[51];
    int number;
    printf("\n\n");
do
{
    printf(" Input Player's Name [3..20]: ");
    scanf("%[^\n"]", name);
    fflush(stdin);
} while(strlen(name)<3 || strlen(name)>20);
printf("\n");
do
{
    printf(" Input Player's Back Number [1..99]: ");
    scanf("%d", &number);
    fflush(stdin);
    exist = false;
    search(&root, number);
    if(exist==true)
    {
        printf("\n * Player's Back Number Cannot be the Same, Please Input Other Number * \n");
        getchar();
        fflush(stdin);
    }
}while(number<1 || number>99 || exist=true);
push(&root, name, number, 0);
getchar();
flush(stdin);
}
a. Declare name and number variable

Name and number will be used to store the data that is inputted by user.

```c
char name[51];
int number;
```

b. Player’s name

Ask user to input player’s name. The length must be between 3 and 20.

```c
do {
    printf(" Input Player's Name [3..20]: ");
    scanf("%[^\n]", name);
    fflush(stdin);
} while(strlen(name)<3 || strlen(name)>20);
```

c. Player’s number

Ask user to input player’s number. The value must be between 1 and 99 and haven’t been exist in the tree.

```c
do {
    printf(" Input Player's Back Number [1..99]: ");
    scanf("%d", &number);
    fflush(stdin);
    exist = false;
    search(&root, number);
    if(exist=true) {
        printf("n " Player's Back Number Cannot be the Same, Please Input Other Number " n");
        getchar();
        fflush(stdin);
    }
} while(number<1 || number>99 || exist=true);
```

d. Push data

Push the data into the tree.

```c
push(&root, name, number, 0);
```

13. Create Menu Method

Menu method will be used to print the available menu.

```c
void menu() {
    printf("n BLUE FOOTBALL MANAGER");
    printf("n 1. View All Player");
    printf("n 2. Add Player");
    printf("n 3. Exit and Pop All");
    printf("n\n >> Input choice : ");
}
```
14. Create Main Method
   
a. Main method will be called first when the program start.

b. In this main method, declare choice variable and ask user to choose menu.

c. The value must be between 1 and 3. Then validate:
   - If user input 1, then call “view” method.
   - If user input 2, then call “addplayer” method.
   - Otherwise, if user input 3, then pop all data from the tree and ends program.
Chapter 07
Binary Search Tree
7.1. **Definition of Binary Search Tree**

**Binary search tree (BST)**, which may sometimes also be called *ordered* or *sorted* binary tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

![A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13](image)

7.2. **Operation of Binary Search Tree**

Operations on a binary search tree require comparisons between nodes. There are several operation in Binary Search Tree such as:

- **Searching**

Searching a binary search tree for a specific value can be a recursive or iterative process.

This explanation covers a recursive method.

- We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree.
- Otherwise, if the value equals the root, the search is successful.
- If the value is less than the root, search the left subtree.
- Similarly, if it is greater than the root, search the right subtree.
This process is repeated until the value is found or the indicated subtree is null.
If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

- Insertion

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value.
In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.
Another way to explain insertion is that in order to insert a new node in the tree, its value is first compared with the value of the root.
If its value is less than the root's, it is then compared with the value of the root's left child.
If its value is greater, it is compared with the root's right child.
This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

- Deletion

There are three possible cases to consider:

- Deleting a leaf (node with no children):
  Deleting a leaf is easy, as we can simply remove it from the tree.

- Deleting a node with one child:
  Remove the node and replace it with its child.

- Deleting a node with two children:
  Call the node to be deleted $N$. Do not delete $N$. Instead, choose either its in-order successor node or its in-order predecessor node, $R$. Replace the value of $N$ with the value of $R$, then delete $R$.
As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree.
In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

7.3. Example

7.1.1. Example 07 – Nokiyem Cellular Phone

Nokiyem Cellular Phone is a famous cellular phone vendor. It has leaded the market for about 7 years. Ms. Iyem as the Vice President of Nokiyem Cellular Phone wants to implement IT technology on her Cellular Phone Product Price List. So she asks you as a skillful programmer to make the program using a binary search tree concept. Here are the descriptions of the prototype:

- The program consists of 4 menus:
  - 1. Add New Phone
  - 2. Update Phone Price
  - 3. Inorder, Preorder, Postorder
  - 4. Exit

- If user chooses Add New Phone (Menu 1), then:
  - Ask user to input phone code. Validate that the phone code must be in this format: N[1-9][0-9][0-9][0-9][0-9]
  - Ask user to input phone price. Validate that the phone price must be between $50 and $999.
  - If the phone code already exists in the tree, then show the message: “--- Add New Phone Failure ---” “Err : Phone Code Already Exists”
  - If tree is still empty, data will be inserted automatically and using BST Concept based on the phone code.
  - Validate that maximum tree level is 4. If level is already at maximum, then show the message “--- Maximum Tree Level is 4 ---”
  - If data has been successfully inputted, show the message “--- Add New Phone Success ---”

- If user chooses Update Phone Price (Menu 2), then:
  - Ask user to input phone code. Validate that the phone code must be in this format: N[1-9][0-9][0-9][0-9][0-9]
  - If data can be found, the program will show the phone’s attributes in this format:
    “Phone Code : [Phone Code]”
    “Phone Price : [Phone Price]”
Then ask user to input a new phone price for the phone code. Validate that the new phone price must be between $50 and $999.

- If data has been successfully changed, then show the message “--- Update Phone Price Success ---”
- If data cannot be found, then show the message “--- Phone Code is Not Found ---”

- If user chooses Inorder, Preorder, Postorder (Menu 3), then:
  - If there is no data in the tree, show the message “--- There is No Phone in The Tree ---”
  - If data is already in the tree, show the attributes of all phones in in-order, pre-order, and post-order.

- If user chooses Exit (Menu 4), then delete all data in the tree and ends the program.
7.1.2. Explanation

a. Create solution named “Soal07.sln” and save to Drive D:

b. Go To main.cpp on Solution Explorer and double click on file

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

c. Type include stdio.h, string.h, and stdlib.h file

d. Declare struct data as digipet data which is have properties on following:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>Integer</td>
<td>Price of phone</td>
</tr>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Number of tree level</td>
</tr>
<tr>
<td>Left</td>
<td>Struct data *</td>
<td>Left subtree</td>
</tr>
<tr>
<td>Right</td>
<td>Struct data *</td>
<td>Right subtree</td>
</tr>
</tbody>
</table>

```
struct data
{
    char code[11];
    int price;
    int level;
    struct data *left, *right;
};
```

e. Declare 1 pointer variable called root.
Each node of a Binary Search Tree (BST) stores a piece of data, each node has bellow it a left subtree and right subtree. The topmost node is called the root and a node with no subtree is called leaf.

```
struct data
{
    char code[11];
    int price;
    int level;
    struct data *left, *right;
} root;
```
f. Create Push Method

Push Method is used for insert a new data into a tree. Data will be sorted by code.

```c
void push(struct data **temp, char code[], int price, int level)
{
    if (level<4)
    {
        if (*temp==NULL)
        {
            *temp = (struct data *)malloc(sizeof(struct data));
            strcpy(*temp->code, code);
            (*temp)->price = price;
            (*temp)->level = level;
            (*temp)->left = (*temp)->right = NULL;

            printf("\n\n\n --- Add New Phone Success ---\n");
        }
        else
        {
            if (strcmp(code, (*temp)->code)<0)
                push(&(*temp)->left, code, price, level+1);
            else if (strcmp(code, (*temp)->code)>0)
                push(&(*temp)->right, code, price, level+1);
            else
            {
                printf("\n\n\n --- Add New Phone Failure ---\n");
                printf(" Err : Phone Code Already Exists\n");
            }
        }
    }
    else
    {
        printf("\n\n\n --- Maximum Tree Level is 4 ---\n");
    }
}
```

Example the tree:

```
                   Root
                     /
                    /  
                N5800   
                  /   
                /     
            N1234   N9550
            /   
           /    
        NULL   NULL
```

```
                      N3510
                    /   
                  /     
                NULL   NULL
```
a. This method takes four parameters. `temp` is passed by reference and the others are passed by value.

```c
void push(struct data **temp, char code[], int price, int level)
```

b. Variable `level` is used to count the number of levels in a tree. In this case, the tree consists of only 4 levels.

```c
if(level<4)
{
}
else
{
    printf("\n\n\n --- Maximum Tree Level is 4 ---\n");
}
```

c. Root variable will be sent at the first time push function called. At the first time, the value of `temp` variable is root.

```c
push(&root, name, job, lvl, damage, 0);
```

d. Insertion data from parameter into a new node.

```c
strcpy(curr->name, name);
strcpy(curr->type, type);
curr->qty = qty;
curr->price = price;
```

e. Check conditions of the tree:

- If node is empty, which is indicated by `(temp == NULL)` condition.
  Insert data from parameter into a new node.

```c
if(*temp==NULL)
{
    *temp = (struct data *)malloc(sizeof(struct data));
    strcpy(*temp->code, code);
    (*temp)->price = price;
    (*temp)->level = level;
    (*temp)->left = (*temp)->right = NULL;
    printf("\n\n\n --- Add New Phone Success ---\n");
}
```
- temp == NULL:

```
root temp
NULL
```

- Insert data parameter into new node, example phone code = "N5800"

```
root temp
N5800
2 left
NULL
right 2
NULL.
```

- If node is not empty, new node will be inserted based on this conditions:

```
else
{
    if(strcmp(code, (*temp)->code)<0)
    {
        push(&(temp)->left, code, price, level+1);
    }
    else if(strcmp(code, (*temp)->code)>0)
    {
        push(&(temp)->right, code, price, level+1);
    }
    else
    {
        printf("\n\n\n--- Add New Phone Failure ---\n");
        printf(" Err : Phone Code Already Exists\n");
    }
}
```

- If the phone code that want to be inserted is smaller than the phone code in temp variable, then the push function will be called again and first parameter on push function is left branch of the temp.

```
if(strcmp(code, (*temp)->code)<0)
{
    push(&(temp)->left, code, price, level+1);
}
```
Temp will move to left branch:

Checks will be repeated again from first step.

Tree after new data successfully inserted:

- If the phone code that want to be inserted is greater than the phone code in temp variable, then the push function will be called again and first parameter on push function is left branch of the temp.

```c
else if(strcmp(code, (*temp)->code)>0)
{
    push(&(*temp)->right, code, price, level+1);
}
```
Temp will move to right branch:

Checks will be repeated again from first step.

Tree after new data successfully inserted:
g. Create Inorder, Postorder and Preorder Method

```c
void InOrder(struct data *temp) {
    if (temp) {
        InOrder(temp->left);
        printf("- %s [ %3d ]\n", temp->code, temp->price);
        InOrder(temp->right);
    }
}

void PreOrder(struct data *temp) {
    if (temp) {
        printf("- %s [ %3d ]\n", temp->code, temp->price);
        PreOrder(temp->left);
        PreOrder(temp->right);
    }
}

void PostOrder(struct data *temp) {
    if (temp) {
        PostOrder(temp->left);
        PostOrder(temp->right);
        printf("- %s [ %3d ]\n", temp->code, temp->price);
    }
}
```

8. Create Pop All Method
Pop All method will delete all node in list

```c
void popall(struct data *temp) {
    if (temp!=NULL) {
        popall(temp->left);
        popall(temp->right);
        free(temp);
    }
}
```
9. Create Menu Method

```c
void menu()
{
    printf("\n NOVITEN CELLULAR PHONE\n");
    printf("\n 1. Add New Phone\n");
    printf("\n 2. Update Phone Price\n");
    printf("\n 3. InOrder, PreOrder, PostOrder\n");
    printf("\n 4. Exit\n");
    printf("\n\n >> Input choice : ");
}
```

10. Create Clear Method
Clear method will clear the screen with enter character ("\n") 25 lines

```c
void clear()
{
    for(int i=0;i<25;i++)
        printf("\n");
}
```

11. Main Method

```c
int main()
{
    int choice;
    do
    {
        clear();
        menu();
        scanf("%d", &choice);
        fflush(stdin);
        switch(choice)
        {
            case 1 : add();
                    break;
            case 2 : update();
                    break;
            case 3 : inprepost();
                    break;
        }
    }while(choice!=4);
    popall(root);
    return 0;
}
```
12. Create menu **Add New Phone** (menu ‘1’)

```c
void add()
{
    char code[11];
    int price;

    printf("\n\n\n");
do {
    printf(" Input Phone Code N[1-9][0-9][0-9][0-9]: ");
    scanf("^[\n]", code);
    fflush(stdin);
} while (codecheck(code) == 0); // call codecheck method
printf("\n");
do {
    printf(" Input Phone Price [$50..$999]: ");
    scanf("$d", &price);
    fflush(stdin);
} while (price < 50 || price > 999);

push( root, code, price, 0 ); // call push method

getchar();
fflush(stdin);
}
```

13. Create codecheck function: 
**The phone code** must be in this format: **N [1-9] [0-9] [0-9] [0-9]**

```c
int codecheck(char text[])
{
    if (strlen(text) != 5) return 0;
    if (text[0] != 'N') return 0;
    if (text[1] < '1' || text[1] > '9') return 0;
    if (text[2] < '0' || text[2] > '9') return 0;
    if (text[3] < '0' || text[3] > '9') return 0;
    if (text[4] < '0' || text[4] > '9') return 0;
    return 1;
}
```

14. Create menu **Update Phone Price** (menu ‘2’)
   a. Declare one global variable

```c
bool found = false;
```
b. Create update method

```c
void update()
{
    char code[11];
    printf("\n\n\n");
    do
    {
        printf(" Input Phone Code N[1-9][0-9][0-9][0-9]: ");
        scanf("%[^\n]", code);
        fflush(stdin);
    } while (codecheck(code)==0);
    found = false;
    searchupdate(&root, code); // call searchupdate method
    if (found==false)
    {
        printf("\n\n\n --- Phone Code is Not Found ---\n");
    }
    else
    {
        printf("\n\n\n --- Update Phone Price Success ---\n");
        getchar();
        fflush(stdin);
    }
}
```

15. Create searchupdate method

```c
void searchupdate(struct data **temp, char code[])
{
    if (*(temp)!=NULL)
    {
        if (strcmp((*temp)->code, code)==0)
        {
            printf("\n Phone Code : %s", (*temp)->code);
            printf("\n Phone Price : %d\n\n", (*temp)->price);
            do
            {
                printf(" Input New Phone Price [$50..$999]: ");
                scanf("%d", &(*temp)->price);
                fflush(stdin);
            } while ( (*temp)->price<50 || (*temp)->price>999);
            found = true;
        }
        searchupdate(&(*temp)->left, code);
        searchupdate(&(*temp)->right, code);
    }
}
```
16. Create menu **InOrder, PostOrder, PreOrder** (menu ‘3’)

```c
void inprepost()
{
    clear();
    if(root==NULL)
    {
        printf(" --- There is No Phone in The Tree ---\n");
    }
    else
    {
        printf("\n Preorder : \n");
        PreOrder(root);
        printf("\n Inorder : \n");
        InOrder(root);
        printf("\n Postorder : \n");
        PostOrder(root);
        printf("\n\n\n");
    }
    getchar();
    fflush(stdin);
}
```

17. If user choose menu **Exit** (menu ‘4’), then call the Pop All Method.

```c
popall(root);
```
Chapter 08

Binary Search Tree

(Parent Node)
8.1. **Operation of Binary Search Tree (continue from chapter 7)**

- **Traversal**
  Once the binary search tree has been created, its elements can be retrieved in-order by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it's recursively accessed. As with all binary trees, one may conduct a pre-order traversal or a post-order traversal, but neither are likely to be useful for binary search trees.

- **Sort**
  A binary search tree can be used to implement a simple but efficient sorting algorithm. Similar to heapsort, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order.

---

8.2. **Example**

8.2.1. **Example 08 - Nokiyem Cellular Phone**

Nokiyem Cellular Phone is a famous cellular phone vendor. It has leaded the market for about 7 years. Ms. Iyem as the Vice President of Nokiyem Cellular Phone wants to implement IT technology on her Cellular Phone Product Price List. So she asks you as a skillful programmer to make the program using a binary search tree concept. Here are the descriptions of the prototype:

- The program consists of 4 menus:
  - 1. Add New Phone
  - 2. Delete Phone
  - 3. Inorder, Preorder, Postorder
  - 4. Exit

- If user chooses **Add New Phone** (Menu 1), then:
  - Ask user to input phone code. Validate that the **phone code must be in this format**: N[1-9][0-9][0-9][0-9]
✓ Ask user to input phone price. Validate that the phone price must be between $50 and $999.
✓ If the phone code already exists in the tree, then show the message:
   “--- Add New Phone Failure ---”
   “Err : Phone Code Already Exists”
✓ If tree is still empty, data will be inserted automatically.
✓ The data is inserted into the tree by using BST Concept based on the phone code.
✓ Maximum tree level is 4. If level is already at maximum, then show the message “--- Maximum Tree Level is 4 ---”
✓ If data has been successfully inputted, show the message: “--- Add New Phone Success ---”
• If user chooses Delete Phone (Menu 2), then:
  ✓ Ask user to input phone code. that the phone code must be in this format:

     N [1-9] [0-9] [0-9] [0-9]

  ✓ If data can be found, the program will delete node and check whether the node has child or not
  ✓ If data has been successfully deleted, then show the message “--- Delete Phone Success ---”
  ✓ If data cannot be found, then show the message “--- Phone Code is Not Found ---”
• If user chooses Inorder, Preorder, Postorder, then:
  ✓ If there is no data in the tree, show the message “--- There is No Phone in The Tree ---”
  ✓ If data is already in the tree, show the attributes of all phones in in-order, pre-order, and post-order.
  ✓ If user chooses Exit, then delete all data in the tree and ends the program
8.2.2. **Explanation**

1. Create solution named “Soal08.sln” and save to Drive D:\

2. Go to main.cpp on solution explorer and double click on file

3. Type include stdio.h, string.h, and stdlib.h file

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

4. Declare struct data as digipet data which is have properties on following:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Char 11</td>
<td>Code of phone</td>
</tr>
<tr>
<td>Price</td>
<td>Integer</td>
<td>Price of phone</td>
</tr>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Number of tree level</td>
</tr>
<tr>
<td>Left</td>
<td>Struct data *</td>
<td>Left subtree</td>
</tr>
<tr>
<td>Right</td>
<td>Struct data *</td>
<td>Right subtree</td>
</tr>
<tr>
<td>Parent</td>
<td>Struct data *</td>
<td>Parent node</td>
</tr>
</tbody>
</table>

```c
struct data
{
    char code[11];
    int price;
    int level;
    struct data *left, *right,*parent;
};
```

5. Declare 1 pointer variable called root.

Each node of a Binary Search Tree (BST) stores a piece of data, each node has bellow it a left subtree and right subtree. The topmost node is called the root and a node with no subtree is called leaf.

```c
struct data
{
    char code[11];
    int price;
    int level;
    struct data *left, *right,*parent;
}; root
```
6. Create Push Method

Push Method is used for insert a new data into a tree. Data will be sorted by code.

```c
void push(struct data **temp, struct data *parent, char code[], int price, int level)
{
    if (level<4)
    {
        if (*temp==NULL)
        {
            *temp = (struct data *)malloc(sizeof(struct data));
            strcpy((*temp)->code, code);
            (*temp)->price = price;
            (*temp)->level = level;
            (*temp)->parent = parent;
            (*temp)->left = (*temp)->right = NULL;
            printf("\a\n --- Add New Phone Success ---\n");
        }
        else
        {
            if (strcmp(code, (*temp)->code)<0)
            {
                push(&(*temp)->left, (*temp), code, price, level+1);
            }
            else if (strcmp(code, (*temp)->code)>0)
            {
                push(&(*temp)->right, (*temp), code, price, level+1);
            }
            else
            {
                printf("\a\n --- Add New Phone Failure ---\n");
                printf(" Err : Phone Code Already Exists\n");
            }
        }
    }
    else
    {
        printf("\n --- Maximum Tree Level is 4 ---\n");
    }
}
```

Example the tree:
a. This method takes five parameters. temp is passing by reference and the others are passing by value.

```c
void push(struct data **temp, struct data *parent, char code[], int price, int level)
```

b. Variable level is used to count the number of levels in a tree. In this case, the tree consists of 4 levels.

```c
if(level<4)
{
}
else
{
    printf("\n\n\n --- Maximum Tree Level is 4 ---\n");
}
```

c. Root variable will be sent at the first time push function called. At the first time, value of temp variable is root. The parent of root is NULL.

```c
push(&root, NULL ,code , price, 0);
```

d. Check conditions of the tree:
   - If node is empty, which is indicated by (temp == NULL) condition. Insert data from parameter into a new node.

```c
if(*temp==NULL)
{
    *temp = (struct data *)malloc(sizeof(struct data)); 1
    strcpy(*temp)->code, code); 1
    (*temp)->price = price;
    (*temp)->level = level; 2
    (*temp)->parent = parent;
    (*temp)->left = (*temp)->right = NULL; 3
    printf("\n\n\n --- Add New Phone Success ---\n");
}
```

   - temp == NULL:
• Insert data parameter into new node, example phone code = "N5800"

```c
else
{
    if(strcmp(code, (*temp)->code)<0)
    {
        push(&(*temp)->left, (*temp), code, price, level+1);
    }
    else if(strcmp(code, (*temp)->code)>0)
    {
        push(&(*temp)->right, (*temp), code, price, level+1);
    }
    else
    {
        printf("\n\n--- Add New Phone Failure ---\n");
        printf(" Err : Phone Code Already Exists\n");
    }
}
```

• If the phone code that want to be inserted is smaller than the phone code in temp variable, then the push function will be called again and first parameter on push function is left branch of the temp.

```c
if(strcmp(code, (*temp)->code)<0)
{
    push(&(*temp)->left, (*temp), code, price, level+1);
}
```
new code = N1234

NULL

parent

N5S000

left

right

push(&(*temp)->left, (*temp), ...);

NULL

 Tempro will move to left branch:

new code = N1234

NULL

parent

N5S000

left

right

temp

NULL

NULL

Checks will be repeated again from first step.

Tree after new data successfully inserted:

new code = N1234

NULL

parent

N5S000

left

right

temp

N1234

left

right

NULL

NULL
7. If the phone code that want to be inserted is greater than the phone code in temp variable, then the push function will be called again and first parameter on push function is left branch of the temp.

```c
else if(strcmp(code, (*temp)->code)>0)
{
    push(&(*temp)->right, (*temp), code, price, level+1);
}
```

**Diagram:**
- New code = N9560
- Temp will move to right branch:
  - New code = N9560
  - Temp will move to right branch:
Checks will be repeated again from first step.

Tree after new data successfully inserted:

```c
void InOrder(struct data *temp)
{
    if(temp)
    {
        InOrder(temp->left);
        printf(" - %s [ $ %3d ]\n", temp->code, temp->price);
        InOrder(temp->right);
    }
}
void PreOrder(struct data *temp)
{
    if(temp)
    {
        printf(" - %s [ $ %3d ]\n", temp->code, temp->price);
        PreOrder(temp->left);
        PreOrder(temp->right);
    }
}
void PostOrder(struct data *temp)
{
    if(temp)
    {
        PostOrder(temp->left);
        PostOrder(temp->right);
        printf(" - %s [ $ %3d ]\n", temp->code, temp->price);
    }
}
```
9. Create Pop All Method

Pop All method will delete all node in list

```c
void popall(struct data *temp)
{
    if(temp!=NULL)
    {
        popall(temp->left);
        popall(temp->right);
        free(temp);
    }
}
```

10. Create Menu Method

```c
void menu()
{
    printf("\n NOKIYEM CELLULAR PHONE");
    printf("\n  "#"="#"="#"="#"="#"="#"="#"="#"="#"="#"="#"="#"="#"="#"="#"");
    printf("\n  1. Add New Phone");
    printf("\n  2. Delete Phone");
    printf("\n  3. InOrder, PreOrder, PostOrder");
    printf("\n  4. Exit");
    printf("\n\n > Input choice :");
}
```

11. Create Clear Method

Clear method will clear the screen with enter character ("\n") 25 lines

```c
void clear()
{
    for(int i=0;i<25;i++)
        printf("\n");
}
```

12. Main Method

Create a simple menu

```c
int main()
{
    int choice;
    do
    {
        clear();
        menu();
        scanf("%d", &choice);
        fflush(stdin);
        switch(choice)
        {
        case 1 : add();
                  break;
        case 2 : deletePhone();
                  break;
        case 3 : inprepost();
                  break;
        }
    }while(choice!=4);
    popall(root);
    return 0;
}
```
13. Create menu **Add New Phone** (menu '1')

```c
void add()
{
    char code[11];
    int price;

    printf("\n\n\n");
    do
    {
        printf(" Input Phone Code N1-9][0-9][0-9][0-9]: ");
        scanf("%\n", code);
        fflush(stdin);
    } while (codecheck(code) == 0); //call codecheck method
    printf("\n");
    do
    {
        printf(" Input Phone Price [$50..$999]: ");
        scanf("%d", &price);
        fflush(stdin);
    } while (price<50 || price>999);

    push(&root, code, price, 0); //call push method

getchar();
    fflush(stdin);
}
```

14. Create codecheck function:

```c
int codecheck(char text[])
{
    if (strlen(text) != 5) return 0;
    if (text[0] != 'N') return 0;
    if (text[1] < '1' || text[1] > '9') return 0;
    if (text[2] < '0' || text[2] > '9') return 0;
    if (text[3] < '0' || text[3] > '9') return 0;
    if (text[4] < '0' || text[4] > '9') return 0;
    return 1;
}
```

The phone code must be in this format: **N [1-9] [0-9] [0-9] [0-9]**
15. Create menu **Delete Phone** (menu ‘2’)

a. Declare one global variable

```c
bool found = false;
```

b. Create deletePhone method

```c
void deletePhone()
{
    char code[11];
    printf("\n\n\n");
    do
    {
        printf(" Input Phone Code N[1-9][0-9][0-9][0-9]: ");
        scanf("%s[\n"]", code);
        fflush(stdin);
    } while (strcmp(code, "") == 0);
    found = false;
    searchDelete(root, code); // call searchDelete method
    if (found == false)
    {
        printf("\n\n\n --- Phone Code is Not Found ---\n");
    }
    else
    {
        printf("\n\n\n --- Delete Phone Code Success ---\n");
    }
    getchar();
    fflush(stdin);
}
```
c. Create searchDelete method

```c
void searchDelete(struct data *temp, char code[])
{
    if (temp != NULL)
    {
        if (strcmp(temp->code, code) == 0)
        {
            if (temp->left == NULL && temp->right == NULL)
            {
                if (temp == root) root = NULL;
                else if (temp->parent != NULL ||
                    temp->parent->left == temp)
                    temp->parent->left = NULL;
                else if (temp->parent->right == temp)
                    temp->parent->right = NULL;
            }
            free(temp);
        }
        else if (temp->left != NULL && temp->right != NULL)
        {
            struct data *largest = rightmost(temp->left);
            strcpy(temp->code, largest->code);
            temp->price = largest->price;
            searchDelete(temp->left, largest->code);
        }
        else
        {
            struct data *child = (temp->left != NULL) ? temp->left : temp->right;
            child->parent = temp->parent;
            if (temp == root) root = child;
            if (temp->parent->left == temp)
                temp->parent->left = child;
            else if (temp->parent->right == temp)
                temp->parent->right = child;
            child->parent = temp->parent;
            free(temp);
        }
    }
    found = true;
}
if (found == false)
    searchDelete(temp->left, code);
if (found == false)
    searchDelete(temp->right, code);
}
```

If data can be found, check conditions of the node:

- If node is a leaf

```c
if (temp->left == NULL && temp->right == NULL)
{
    if (temp == root) root = NULL;
    else if (temp != root){
        if (temp->parent->left == temp)
            temp->parent->left = NULL;
        else if (temp->parent->right == temp)
            temp->parent->right = NULL;
    }
    free(temp);
}
```
Check whether the node is root or not.
Example: delete N3510

In this case, N3510 is right child of N1234. This code will be executed:

```c
else if( temp->parent->right == temp )
    temp->parent->right = NULL;
```

```c
temp->parent->right = NULL;
```
If node has 2 child:

```c
else if ( temp->left!=NULL && temp->right!=NULL )
{
    struct data *largest = right_most(temp->left);
    strcpy(temp->code, largest->code);
    temp->price = largest->price;
    searchDelete(temp->left, largest->code);
}
```
Example: delete \textbf{N1234}

Call right\_most method to find the largest on the left node:

```c
struct data *right_most(struct data *curr) {
    if (curr->right == NULL) return curr;
    return right_most(curr->right);
}
```
Change the **temp** data into **largest** data:

```c
strcpy(temp->code, largest->code);
temp->price = largest->price;
```

Call **searchDelete** method to delete N3456 on the left of **temp**:

```c
searchDelete(temp->left, largest->code);
```
If node has 1 child, check whether the child of node is on the left or right

```c
else {
    struct data *child = (temp->left != NULL) ? temp->left : temp->right;
    child->parent = temp->parent;
    if (temp==root) root=child;
    if(temp->parent->left == temp)
        temp->parent->left = child;
    else if(temp->parent->right == temp )
        temp->parent->right = child;
    child->parent = temp->parent;
    free(temp);
}
```

Example: delete **N1234**

In this case, child of node is on the right node.
child->parent = temp->parent;

free(temp);
16. Create menu **InOrder, PostOrder, PreOrder** (menu ‘3’)

```c
void inprepost()
{
    clear();
    if(root==NULL)
    {
        printf(" --- There is No Phone in The Tree ---\n");
    }
    else
    {
        printf("\n Preorder : \n");
        PreOrder(root);
        printf("\n Inorder : \n");
        InOrder(root);
        printf("\n Postorder : \n");
        PostOrder(root);
        printf("\n\n\n");
    }
    getchar();
    fflush(stdin);
}
```

17. If user choose menu **Exit** (menu ‘4’), then call the Pop All Method.

```c
popall(root);
```
Chapter 09

AVL Tree
9.1. **Definition of AVL Tree**

The AVL tree is named after its two Soviet inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper “An algorithm for the organization of information.”

AVL Tree is an improvement from Binary Search Tree. Assume that the data inputted in Binary Search Tree are 3, 5, 7, 9, 10. Then, Binary Search Tree will look like:

![Binary Search Tree Example](image1)

In that case, Binary Tree will not effective. If we want to find node with value 10, all the nodes must be passed through. In AVL Tree, this case will be eliminated because it will make balance between left and right side. The heights of the two child subtrees of any node differ by at most one. The AVL Tree will look like:

![AVL Tree Example](image2)

A technic named rotation is used to rebalance this tree when inserting or deleting. Left rotation is used when the height of left subtrees differs by 2 with the height of right subtrees, vice versa. Sometimes, double rotation is used in specific case.

The rotation technic will be described below:

- Nodes with value of 3, 5, 10 are inserted. So, the tree is imbalanced. Left rotation is used to rebalance the tree.

![Rotation Example](image3)
- Nodes with value 8, 12, 15 are inserted after 3, 5, 10. The tree is imbalanced in node with node 5 because the height of left subtree differs by 2 with the height of the right subtree. Left rotation is again used to rebalance the tree.

- Nodes with value 20 and 17 are inserted.
The tree is imbalanced at node with value 15. Double rotation is used.
  First, right rotation at node 20, then, left rotation at node 15.

- Node 16 is inserted. The tree is imbalanced at node 12. Double rotation is used.
9.2. Example

9.2.1. Example 09 – AVL Tree

- The program consists of 3 menus:
  - 1. Insert data
  - 2. Show tree
  - 3. Exit

- If user chooses Insert data (Menu 1), then:
  - Ask user to input number. Validate that the number must be between 1 and 1000.
  - If the number already exists in the tree, then show the message: “There is already data with the same value”

- If user chooses Show tree (Menu 2), then:
  - If there is no data in the tree, show the message “There is No Data”
  - If data is already in the tree, show all numbers in simple tree format.

- If user chooses Exit (Menu 3), then delete all data in the tree and ends the program.

9.2.2. Explanation

1. Create solution “Answer09.sln” and save to Drive D:\
2. Go To Answer09.cpp on Solution Explorer and double click on file

3. Type include stdio.h, string.h, and stdlib.h file

```c
#include <stdio.h>
#include <stdlib.h>
```

4. Declare struct data with the following properties:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>int</td>
<td>Node value</td>
</tr>
<tr>
<td>Height</td>
<td>int</td>
<td>node</td>
</tr>
<tr>
<td>left</td>
<td>struct data *</td>
<td>Left subtree</td>
</tr>
<tr>
<td>right</td>
<td>struct data *</td>
<td>Right subtree</td>
</tr>
<tr>
<td>parent</td>
<td>struct data *</td>
<td>Parent node</td>
</tr>
</tbody>
</table>
5. Declare 1 pointer variable called root.
   Each node of a AVL Tree stores a piece of data, each node has below it a left subtree and right subtree. The topmost node is called the root and a node with no subtree is called leaf.

   ```
   struct data{
       int value, height;
       struct data *parent, *left, *right;
   } *root = NULL;
   ```

6. Create Max Method
   Max method will return the biggest number between 2 numbers.

   ```
   int max(int a, int b) {
       if (a > b) return a;
       return b;
   }
   ```

7. Create Clear and Space Method
   Clear method will clear the screen and space method will print spaces.

   ```
   void clear() {
       int i;
       for (i=0; i<25; i++) printf ("\n");
   }

   void space(int n) {
       int i;
       for (i=0; i<n; i++) printf (" ");
   }
   ```

8. Create Get_Height Method
   Get_Height method will return node height or 0 (if NULL).

   ```
   int get_height(struct data *curr){
       if (curr==NULL) return 0;
       return curr->height;
   }
   ```

9. Create Swap_Parent Method
   Swap_Parent method will swap parent of two nodes (parent and child). The child can be either left or right child.

   ```
   void swap_parent(struct data *c, struct data *p){
       c->parent = p->parent;
       if (p == root) root = c;
       else if (p->parent->left == p) p->parent->left = c;
       else if (p->parent->right == p) p->parent->right = c;
       p->parent = c;
   }
   ```
Here is the illustration how it works:

In the illustration, node p is located exactly under node c, because node p can be either left or right child of node c depends on what kind of rotates will execute whether left or right rotation.

10. Create Rotate_Right and Rotate_Left Method

**Rotate_Right Method**

```
void rotate_right(struct data *lc, struct data *p){
    swap_parent(lc, p);
    p->left = lc->right;
    if(p->left != NULL) p->left->parent = p;
    lc->right = p;
    p->height = max(get_height(p->left), get_height(p->right)) + 1;
    lc->height = max(get_height(lc->left), get_height(lc->right)) + 1;
}
```

**Rotate_Left Method**

```
void rotate_left(struct data *rc, struct data *p){
    swap_parent(rc, p);
    p->right = rc->left;
    if(p->right != NULL) p->right->parent = p;
    rc->left = p;
    p->height = max(get_height(p->left), get_height(p->right)) + 1;
    rc->height = max(get_height(rc->left), get_height(rc->right)) + 1;
}
```

Here is the illustration how rotate_left works:
11. Create New_Node Method

New_Node method will allocate memory for new node and set its value.

```c
struct data* newnode(int number, struct data *parent)
{
    struct data *node = (struct data*) malloc(sizeof(struct data));
    // memory allocation
    node->value = number;
    node->height = 1;
    node->parent = parent;
    node->left = node->right = NULL;
    printf("\nNode inserted...");
    return node;
}
```

12. Create Insert Method
   
a. Create Root Node

Each data that enter the tree will be normally inserted, similar with inserting into binary search tree. After inserted, the node will be checked if there is an unbalanced node. If there are any, then a balancing action will be taken.

```c
void insert(int number){
    if(root == NULL) root = newnode(number, root);
    else
    {
        // insert new node into the tree (2)
        // balance tree (3)
    }
}
```

b. Insert New Node Into the Tree

New node will be inserted into the tree. This method is similar with inserting node to binary search tree.

```c
int insert = 0;
struct data *curr = root;
while(!insert){
    if(number < curr->value) {
        if(curr->left == NULL) {
            curr->left = newnode(number, curr); insert = 1;
        }
        curr = curr->left;
    }
    else if(number > curr->value) {
        if(curr->right == NULL) {
            curr->right = newnode(number, curr); insert = 1;
        }
        curr = curr->right;
    }
    else {
        printf("There is already data with the same value\n");
        return;
    }
}
```
c. Balance the Tree

```c
while(curr != root){
    if(curr->parent->left == curr){
        int l = curr->height;
        int r = get_height(curr->parent->right);
        curr->parent->height = max(l, r)+1;
        if(l - r > 1){
            if(number < curr->value) rotate_right(curr, curr->parent);
            else {
                rotate_left(curr->right, curr);
                rotate_right(curr->parent, curr->parent->parent);
            }
        }
    } else if(curr->parent->right == curr){
        int l = get_height(curr->parent->left);
        int r = curr->height;
        curr->parent->height = max(l, r)+1;
        if(r - l > 1){
            if(number > curr->value) rotate_left(curr, curr->parent);
            else {
                rotate_right(curr->left, curr);
                rotate_left(curr->parent, curr->parent->parent);
            }
        }
    }
    return;
}
curr = curr->parent;
```

After inserting, the ascendant nodes will update height. If there is an ascendant node that is imbalanced, then rotation will be done.

Assume that a number (n) is inserted into tree. Curr pointer (c) will go up to root and update height.

An imbalanced node is found (in orange color), then rotation will be done.

![Diagram of tree rotation](image)
Assume that a number (n) is inserted into the tree. Curr pointer (c) will go up to the root and update height.

An imbalanced node is found (in orange color). (the difference is now c<n).

Illustration of double rotation:

Because c < n, then the type of rotation is double. First, left_rotate (c->r, c), then right_rotate(c->p, c->p->p), then the tree now is balanced.

13. Create View Method

View method will show tree nodes in a tree structure.

```c
void view(struct data *curr, int level){
    space((level - 1)*2);
    if(curr==NULL){ printf( "--NULL\n"); return; }
    else printf("--%d\n", curr->value);
    view (curr->left, level+1);
    view (curr->right, level+1);
}
```

14. Create Popall Method

Popall method will delete all nodes in the tree.

```c
void popall(struct data *curr){
    if(curr!=NULL){
        popall(curr->left);
        popall(curr->right);
        free(curr);
    }
}
```
15. Main Method

```c
int main()
{
    int choose, number;
    printf("AVL TREE\n");
    printf("=========\n");
    do{
        clear();
        printf("1. Insert data\n");
        printf("2. Show Tree\n");
        printf("3. Exit\n");
        printf("What do you want to choose : ");
        scanf("%d", &choose);
        fflush(stdin); printf("\n");
        switch(choose){
            case 1:
                break;
            case 2:
                break;
        }
        getchar();
    } while(choose!= 3);
    popall(root);
    return 0;
}
```

16. Create Insert Data Menu

```c
case 1:
    do{
        printf("Insert number [1..1000] : ");
        scanf("%d", &number); fflush(stdin);
    } while(number<1 || number > 1000);
    insert (number);
    break;
```

17. Create View Menu

```c
case 2:
    if(root== NULL) printf("There is no data\n");
    else view(root, 1);
    break;
```
Chapter 10

HEAP
10.1. Definition of HEAP

HEAP is a specialized tree-based data structure that satisfies the heap property: if “B” is a child node of “A”, then key(“A”) ≤ key(“B”).

This implies that an element with the smallest key is always in the root node, and so such a heap is sometimes called a min-heap. (Alternatively, if the comparison is reversed, the greatest element is always in the root node, which results in a max-heap.)

The several variants of heaps are the prototypical most efficient implementations of the abstract data type priority queues. Besides that, a heap is a complete binary tree, which leads to the idea of storing it using an array. By utilizing array-based representation, we can reduce memory costs while tree navigation remains quite simple.

A heap is stored in array level by level. The topmost level contains root only. It is mapped to the first element of an array (with index 0).

Root's children are mapped to the second and third elements and so on. A heap is a complete binary tree, which guarantees, that heap's nodes take up places in the array compactly, making the mapping quite efficient.

Assume that the data inputted in Heap are 1, 5, 10, 13, 15, 18 and 11. Then, the heap will look like

![Heap Diagram]

Mapping formula:

Left(i) = 2 * n + 1
Right(i) = 2 * n + 2

for the example: 2*0 + 1 = 1 value in index 1 is 5
for the example: 2*0 + 2 = 2 value in index 2 is 10

Parent(i) = (n - 1) / 2
10.2. Example

10.2.1. Example 10 - HEAP

The program simulates an HEAP.

- The program consists of 3 menus:
  - 1. Insert data
  - 2. View data
  - 3. Delete root
  - 4. Exit

- If user chooses Insert data (Menu 1), then:
  - Ask user to input number. Validate that the number must be between 1 and 99.

- If user chooses View All Data (Menu 2), then:
  - If there is no data in the tree, show the message “There is No Data”
  - If data is already in the tree, show all numbers in Tree.

- If user chooses Delete root (Menu 3), then
  - If there is no data in the tree, show the message “There is No Data”
  - If data is already in the tree, delete root

- If user chooses Exit (Menu 4), then the program will ends

10.2.2. Explanation

1. Create solution named “Answer10.sln” and save to Drive D:\

2. Go To Answer09.cpp on Solution Explorer and double click on file

   ![Solution 'Answer10' (1 project)]
   - Anwer10
     - Header Files
     - Resource Files
     - Source Files
     - C++ Anwer10.cpp

3. Type include stdio.h

   ```
   #include <stdio.h>
   ```

4. Declare a new array of integer and int

   ```
   int DATA[50];
   int INDEX = 1;
   ```
5. Create clearscreen method

Clearscreen method will clear the screen

```c
void refresh(){
    for(int i=0;i<30;i++)
    {
        printf("\n");
    }
}
```

6. Create swap method

```c
void swap(int *Data1, int *Data2) {
    int temp_Data = *Data1;
    *Data1 = *Data2;
    *Data2 = temp_Data;
}
```

7. Create push method

```c
void push(int new_data) {
    int curr = INDEX;
    DATA[curr] = new_data;
    while ( curr != 1 & & DATA[curr/2] > DATA[curr] ) {
        int parent = curr / 2;
        if ( DATA[parent] <= DATA[curr] ) break;
        swap(&DATA[curr],&DATA[parent]);
        curr = parent;
    }
    INDEX++;
    printf("Insert success");
    getchar();
}
```

If the tree still empty, it will insert data into index 0 or root. Else if it has another data and new_data is smaller than another data in tree, then the new_data will go to lastest index.
Compare 3 and 7

Because 3 is more smaller than 7, swap 3 and 7

Compare again 3 and 5

3 is more smaller than 5 (root), then 3 will become root and 5 will replace 3’s place
8. Create method print_data
   Method print_data will print all data in Tree, because INDEX is start from 1, then we will subtraction it with 1

```c
void print_data(){
    if(INDEX==1){
        printf("There is no data");
        getchar();
    }
    else{
        for(int i=1;i<INDEX;i++)
        {
            printf("Index : %d, Data : %d \n",i-1,DATA[i]);
        }
        getchar();
    }
}
```
9. Create pop method

```c
void pop() {
    if (INDEX == 1) {
        printf("There is no data");
        getchar();
    } else {
        int curr = 1, LEFT, RIGHT;
        INDEX--;
        swap(&DATA[1], &DATA[INDEX]);
        while (curr * 2 < INDEX) {
            LEFT = curr * 2;
            RIGHT = curr * 2 + 1;
            if (DATA[LEFT] < DATA[curr]) {
                if (RIGHT < INDEX && DATA[RIGHT] < DATA[LEFT]) {
                    swap(&DATA[curr], &DATA[RIGHT]);
                    curr = RIGHT;
                } else {
                    swap(&DATA[curr], &DATA[LEFT]);
                    curr = LEFT;
                }
            } else if (RIGHT < INDEX && DATA[RIGHT] < DATA[curr]) {
                swap(&DATA[curr], &DATA[RIGHT]);
                curr = RIGHT;
            } else break;
        }
        printf("Delete success");
        getchar();
    }
}
```

Delete Root

For the example, if we want to delete root

```
     0
   /   
  5     10
 /   \\ / \\
18 11 12 13
```

The lastest index [6] will become root

\[
\begin{array}{c}
\text{swap(data[0], data[6]);}
\end{array}
\]

Remember, root’s node must smallest than others, and current root (11) is more larger than it child (5), so the smallest value (5) in tree will replace current root’s place

\[
\begin{array}{c}
\text{DATA[LEFT] < DATA[curr]}
\text{5 < 11}
\end{array}
\]

\[
\begin{array}{c}
\text{swap(&data[0],&DATA[1]);}
\text{curr = LEFT;}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{index}
5 & 11 & 10 & 13 & 15 & 18
\end{array}
\end{array}
\]
Remember, in max heap, the greatest value will become root INDEX's value must be subtraction it with 1 because we’ve declared INDEX =1.

10. Create Main method

```c
void main(){
    int input, data;
    do{
        refresh();
        print_menu();
        scanf("%d", &input);fflush(stdin);
        switch(input)
        {   
          case 1 : menu_insert();break;
          case 2 : print_data();break;
          case 3 : pop();break;
        }
    }while(input!=4);
}
```

11. Create menu_insert

```c
void menu_insert(){
    int data;
    do{
        printf("Input your data [1..99] ");scanf("%d", &data);fflush(stdin);
    }while(data<1 || data>99);
    push(data);
}