CS-101 Programming Fundamentals

Lab Manual

Habib University

First printing, July 2018

Contents

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

0.1 Citation

This statement requires citation [**article_key**]; this one is more specific [**book_key**].

Week 1

1. Variables, Loops, Conditionals and functions

1.1 Objective

In this lab, we'll learn the definition and assignment of variables. We will also learn the implementation and usage of loops and conditionals. Moreover, we will also learn the use of functions.

1.2 Description

1.2.1 Variables

Variables allows to store data to be used in program. In C++, the variable must be declared with it's data type. Once declared, data of the declared data type can be assigned to the variable (See Example 1.1)
The table below shows size, ranges and the syntax in C++ of some data types.

Data type	Size (in bytes)	Range	Keyword
Character	1	-128 to 127	char
Integer	4	-2147483648 to +2147483647	int
Boolean	1 bit	true/false	bool
Floating point	4	-3.4e-38 to +3.4e-38	float
Double floating point	8	1.7e-308 to 1.7 e+308	double

Table 1.1: Properties of Data Types

Example 1.1

```
#include <iostream>
int main()
{
    //declaring integers
    int number1;
    number1 = 4;
    int number2 = 5; //alternative way

    //Declaring character
    char character1 = 'a';

    //Declaring bool
    bool boolean1 = true;
    bool boolean2 = false;

    //Declaring float
    float fpoint = 1.777;

    //Declaring double
    double dpoint = 4.9999997778787868685767689 9;
    return 0;
}
```

Declaring Variables

■

Allocation of memory

Computer's memory can be viewed as a series of cubbyholes. Each cubbyhole is one of many such holes all lined up. Each cubbyhole or memory location is numbered sequentially. These numbers are known as memory addresses. A variable reserves one or more addresses in which a binary value is stored. Each address is mostly one byte (8 bits large)

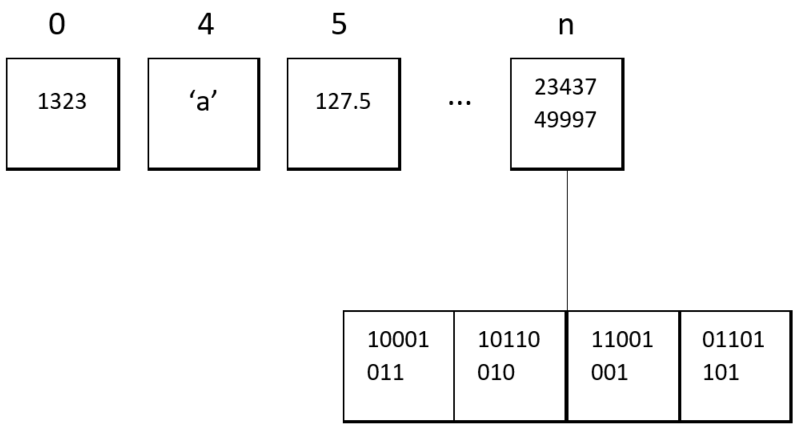


Figure 1.1: Memory Allocation of various data types

In Figure 1.1, the integer '1323' has taken 4 cubby holes or blocks sequentially from address '0' to '3'. Since a character is of 1 byte, character 'a' has taken only one cubby hole of address '4'. The integer '234749997' has taken 4 cubby holes. Each cubby hole is of 8 bits or 1 byte. Therefore the binary representation of the integer is stored sequentially in the cubby holes.

1.2.2 Conditional statements

Conditional statements are used to make decisions based on a given condition.

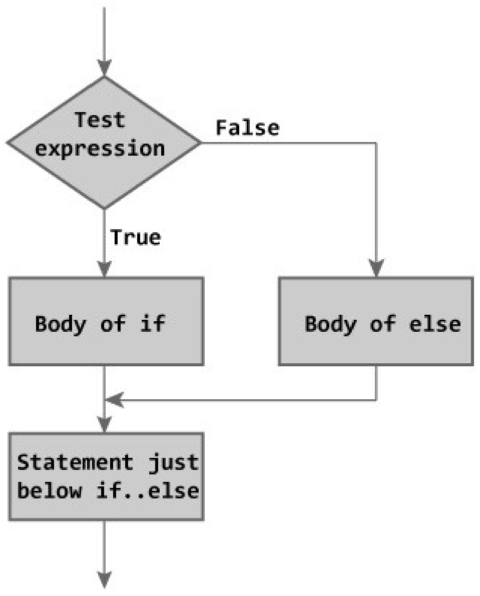


Figure 1.2: Flowchart for if-else conditionals

Comparison operators with truth value

Conditional statements works on the basis of the truth value of the expressions used. There are multiple comparison operators which returns a truth value in boolean and are listed below.

Operator	Description	Usage Example
$==$	equal to	$1 == 1 \rightarrow$ true 't' $==$ 'p' $\rightarrow$ false
$!=$	not equal to	$5! = 5 \rightarrow$ false $4.6! = 6.7 \rightarrow$ true
$<$	less than	$5 < 4 \rightarrow$ false 'a' $<$ 'b' $\rightarrow$ true (using ASCII value)
$<=$	less than or equal to	$4.5 <= 4.5 \rightarrow$ true $3 <= 5.7 \rightarrow$ true
$>=$	greater than or equal to	'b' $>=$ 'b' $\rightarrow$ true true $>=$ false $\rightarrow$ true

Table 1.2: Comparison operators

R Do not get confused between the assignment operator and equal to operator when using in an if conditional statement

The example below shows an example usage of if conditionals and comparison operators.

■ **Example 1.2**

```
#include <iostream>
using namespace std;
int main()
{
int num; //variable to store input
cout<<"Enter number: "<<endl; //To show prompt
cin>>num; //Taking and storing input value
if(num % 2 == 0) //"a%b" to check the remainder when 'a' is  divided by 'b'
{
cout<<"Number is divisible by 2"<<endl;
}
else
{
cout<<"Number is not divisible by 2"<<endl;
}
return 0;
}
```

Using if condition to check if number is divisible by 2

■

1.2.3 Loops

Loop is a sequence of instructions being repeated until a specific condition or target is reached.

There are three types of loops in C++:

While loop

A while loop statement repeatedly executes a target statement as long as a given condition is true. The syntax of a while loop loop in C++ is:

```
while (testExpression)
{
    statement(s);
}
```

where, **testExpression** is checked on each entry of the while loop.

Here is the flow of control in a while loop:

- The while loop evaluates the **test expression**. If the **test expression** is true, codes inside the body of while loop is evaluated.
- Then, the **test expression** is evaluated again. This process goes on until the **test expression** is false.
- When the **test expression** is false, while loop is terminated.

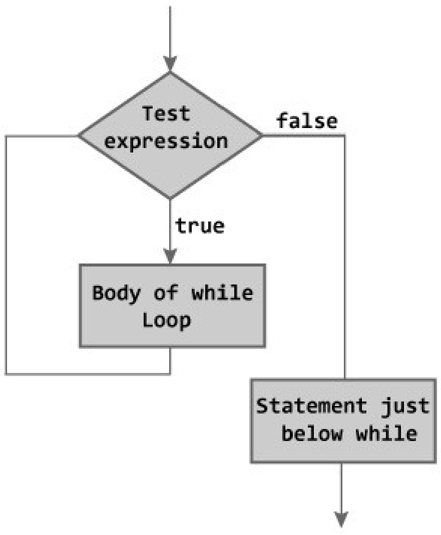


Figure 1.3: Flowchart of while loop

■ **Example 1.3**

```
#include <iostream>
int main()
{
    int num = 1; //variable to be incremented
    int count = 0; //to keep track of number of iterations
    while(count < 5)
    {
        num = num + 1; //or num++ can also be used to increment
        count++;
    }
    return 0;
}
```

Using While loop to increment an integer 5 times

■

Do-while loop

The syntax of do..while loop is:

```
do {
    // codes;
}
while (testExpression);
```

Here is the flow of control in a while loop:

- The codes inside the body of loop is executed at least once. Then, only the **test expression** is checked.
- If the **test expression** is true, the body of loop is executed. This process continues until the **test expression** becomes false.
- When the **test expression** is false, do...while loop is terminated.

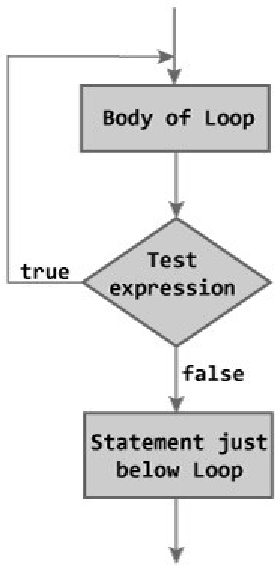


Figure 1.4: Flowchart of do-while loop

■ **Example 1.4**

```
#include <iostream>
int main()
{
    int num = 1; //variable to be incremented
    int count = 0; //to keep track of number of iterations
    do
    {
        num = num + 1; //or num++ can also be used to increment
        count++;
    }
    while(count<5)
    return 0;
}
```

Using While loop to increment an integer 5 times

■

For loop

A for loop allows you to efficiently write a loop that needs to execute a specific number of times. The syntax of a for loop in C++ is:

```
for(initializationStatement; testExpression; updateStatement)
{
    statement(s);
}
```

Here is the flow of control in a for loop:

- The **initialization statement** is executed only once at the beginning.
- Then, the **test expression** is evaluated. If the **test expression** is false, for loop is terminated. But if the **test expression** is true, codes inside body of for loop is executed and **update statement** is executed.
- Again, the **test expression** is evaluated and this process repeats until the **test expression** is false.

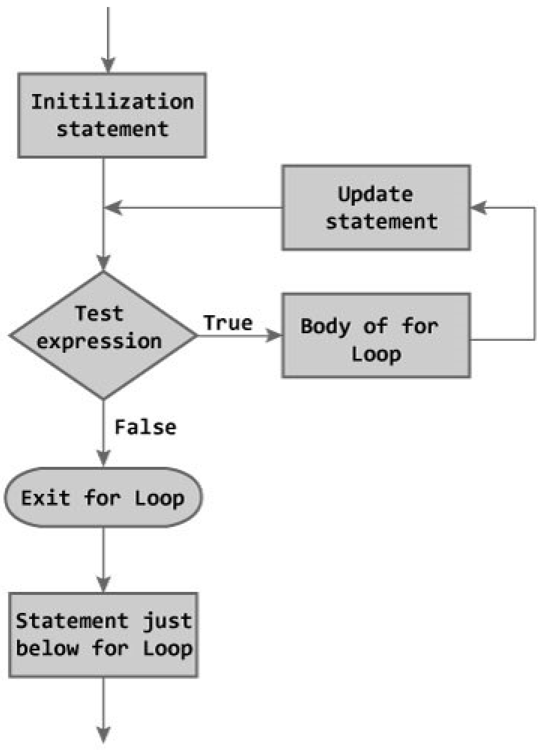


Figure 1.5: Flowchart of for loop

■ Example 1.5

```
#include <iostream>
int main()
{
    int num = 1; //variable to be incremented
    for(int i = 0; i < 5; i++)
    {
        num = num + 1; //or num++ can also be used to increment
    }
    return 0;
}
```

Using For loop to increment an integer 5 times

■

■ Example 1.6

```
#include <iostream>
int main()
{
    int num = 1; //variable to be incremented
    int count = 0; //to keep track of number of iterations
    while(count < 5)
    {
        num = num + 1; //or num++ can also be used to increment
        count++;
    }
    return 0;
}
```

Using While loop to increment an integer 5 times

■

1.2.4 Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Defining a Function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter1 , parameter2 , parameter3 ,... )
{
    body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type** - A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

- **Function Name** - This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters** - A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body** - The function body contains a collection of statements that define what the function does.

Declaring a function

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

```
return_type function_name( parameter list );
```

Calling a function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

■ **Example 1.7**

```
#include <iostream>
using namespace std;

// function declaration
int max(int num1, int num2);

int main ()
{
    // local variable declaration:
    int a = 100;
    int b = 200;
    int ret;

    // calling a function to get max value.
    ret = max(a, b);
    cout << "Max value is : " << ret << endl;

    return 0;
}

// function returning the max between two numbers
int max(int num1, int num2)
{
    // local variable declaration
    int result;

    if (num1 > num2)
    result = num1;
    else
    result = num2;

    return result;
}
```

Declaration, definition amd calling of a max function

■

1.3 Problems

Problem 1.1 A certain grade of steel is graded according to the following conditions:

- Hardness must be greater than 50
- Carbon content must be less than 0.7
- Tensile strength must be greater than 5600

The grades are as follows:

- Grade is 10 if all three conditions are met
- Grade is 9 if conditions (i) and (ii) are met
- Grade is 8 if conditions (ii) and (iii) are met
- Grade is 7 if conditions (i) and (iii) are met
- Grade is 6 if only one condition is met
- Grade is 5 if none of the conditions are met

Write a program, which will require the user to give values of hardness, carbon content and tensile strength of the steel under consideration and output the grade of the steel.

Instructions

- Use if-else conditional statements

Problem 1.2 Two frogs named 'FrogPrime' and 'Frogatron' are in a well of depth 1000 feet. They decide to race each other to the top to see who is the winner.

Both frogs can jump 4 feet at a time but slide down 1 foot every time, thus making the total distance covered to be 3 feet.

FrogPrime has a 2 % chance to get an adrenaline rush and jump 5 feet. Frogatron has special claws that have a 2% chance to grab on to a wall, thus it does not slide down 1 foot if the claws connect

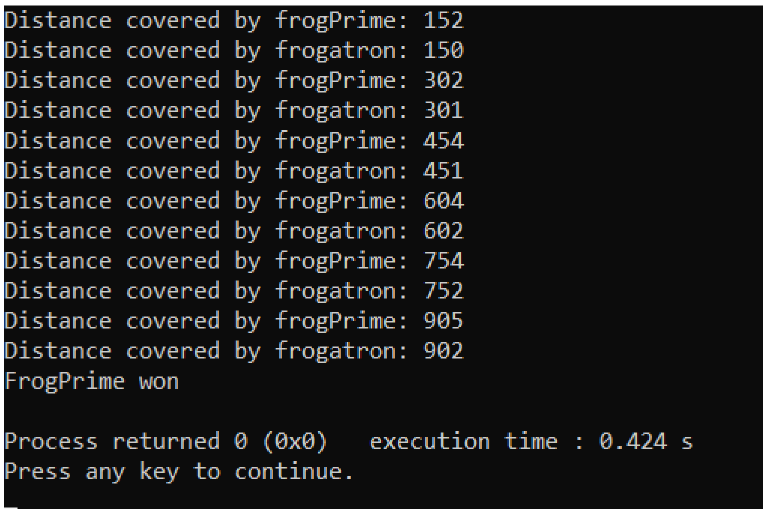
The frogs will keep on jumping till one of them clears the well and is declared a winner.

Your program should simulate this behavior. It should find out the winner and report it. It should also report if there is a tie.

It should also display the progress of the frogs at every 50th jump.

It should also display the total number of jumps once the competition is over.

The output should be similar to this:



```
Distance covered by frogPrime: 152
Distance covered by frogatron: 150
Distance covered by frogPrime: 302
Distance covered by frogatron: 301
Distance covered by frogPrime: 454
Distance covered by frogatron: 451
Distance covered by frogPrime: 604
Distance covered by frogatron: 602
Distance covered by frogPrime: 754
Distance covered by frogatron: 752
Distance covered by frogPrime: 905
Distance covered by frogatron: 902
FrogPrime won

Process returned 0 (0x0)   execution time : 0.424 s
Press any key to continue.
```

Instructions

- You can use random number generation to simulate chances. Find out yourself about using random number generation and ranges.
- Think in terms of loops and if-else conditionals

Problem 1.3 Write a function IsPrime(int n) to test whether a parameter is prime. Apply the function in a program which prints all the prime numbers up to 100.

Instructions

- Use modulo operator to check divisibility
- You can use bool to keep check if a number is prime
- Break the question in parts and work on the function first

Problem 1.4 Write a function Fibonacci(int n) which calculates the nth Fibonacci number. Write a main program that uses this function and the function from Problem 1.3 to print out the first 5 Fibonacci numbers that are also primes.

Instructions

- Think in terms of loops and an extensive use of variables to keep track of the needed terms in every iteration

1.4 Feedback

Please write the things you've learned from this lab and suggestions to make it more better and easy to learn.

2. Stacks

2.1 Objective

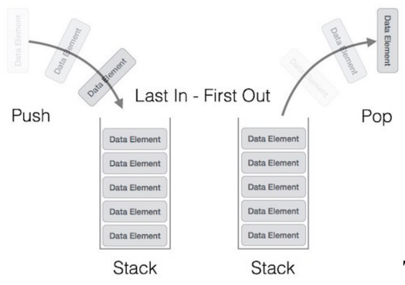
In this lab, we'll learn about a data structure, called Stack. We'll learn about its implementation, operations and uses.

2.2 Description

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push**: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop**: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top**: Returns top element of stack.
- **isEmpty**: Returns true if stack is empty, else false.



There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Figure 2.1: Representation of stack

2.2.1 Real-life example of stack

2.2.2 Time Complexities of operations on stack

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

2.2.3 Applications of stack

There are many possible applications of stack. Some are listed below:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver

2.2.4 Implementation

We will be implementing a linked list based stack. In a linked list based stack, we have nodes linked to each other. Each node stores a data and a link to the node next to it.

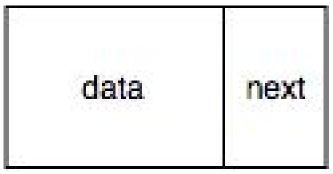


Figure 2.2: Representation of node

Implementing Node

To implement node, we will be making a struct to hold data and the link. In this implementation we are making a node to store integer type data. However, it can be implemented to store any data type or object.

```
struct Node
{
    int data; \\To store data
    Node* next; \\To store the address or link to the next node
};
```

Initializing the Stack

As a stack consists of many nodes, we should have many nodes inside our stack class, but we will have only one node, "head", which will act as a reference node and allows us to iterate through all the nodes till the end of the stack.

```
class Stack
{
    private:
        Node* head;

    public:
        Stack()
        {
            head = NULL;
        }
};
```

Our stack now consists of a Node pointer which will hold the address of the head (i.e the top element) of the stack. To initialize, it is set to **NULL**, as the stack is empty.

Figure 2.3: Stack's head pointer

Implementing Push() operation

Inside the class of Stack, we will declare and define a function "Push(int val)" in public.

```
void Push(int val)
{
    if (head == NULL)
    {
        head = new Node;
        head->data = val;
        head->next = NULL;
    }
    else
    {
        Node* temp = head;
        head = new Node;
        head->data = val;
        head->next = temp;
    }
}
```

■ **Example 2.1** Let’s visualize an example Push() operation

```
Stack stack;
stack.Push(5);
```

Since the stack is empty and head is set to NULL. Body of ’if’ will be executed.
It first creates a new node

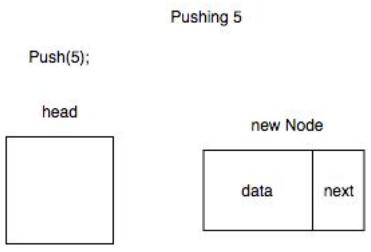


Figure 2.4: Creating new node for Push(5)

Then it sets head pointer to the new node.

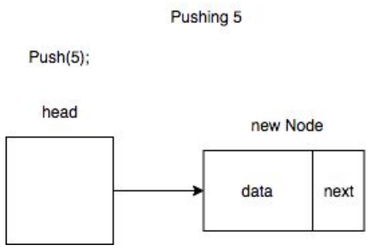


Figure 2.5: Setting head pointer

Then it sets the data and next pointer of new node.

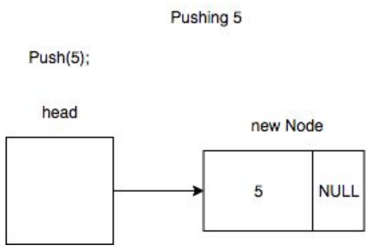


Figure 2.6: Setting data and next pointer

■

Since, head is not **NULL**, Push operation will now execute body of 'else'. Let's see an example

■ **Example 2.2**

```
stack.Push(3);
```

It will first create a temporary node pointer, temp.

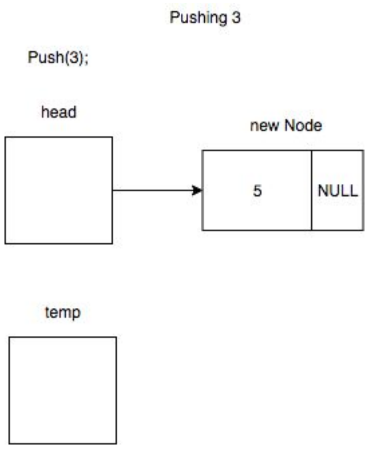


Figure 2.7: Creating temp pointer

Then it sets the temp to the node where head is pointing

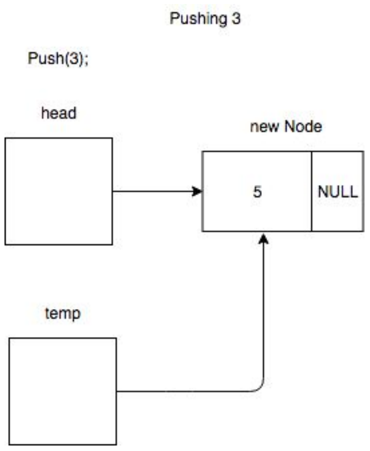


Figure 2.8: Setting temp node

It now creates a new node and sets head to the newly created node.

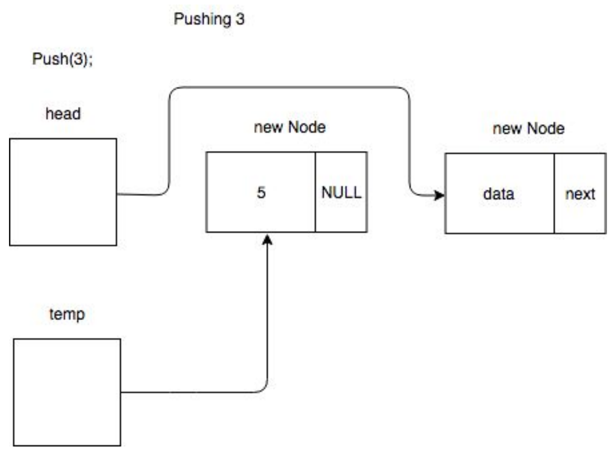


Figure 2.9: Setting head to the new node

It sets the newly created node's next pointer to the node being pointed by temp.

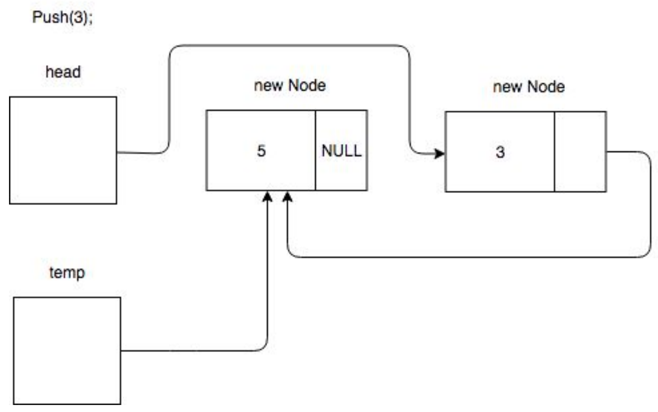


Figure 2.10: Setting next pointer

The temp gets destroyed since it was a local variable of the function and 3 is added succesfully.

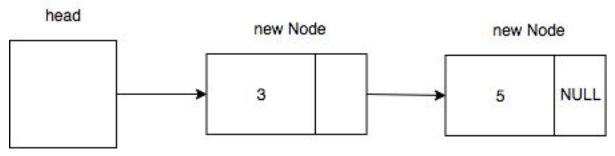


Figure 2.11: Succesful Push(3) operation

■

Implementing Pop() Operation

Inside the class of Stack, we will declare and define a function "Pop()" in public.

```
int Pop()
{
    int val = -1;
    if(head != NULL)
    {
        Node* temp = head;
        head = head->next;
        val = temp->data;
        delete temp;
        temp = NULL;
    }
    return val;
}
```

Let's visualize an example of Pop() operation

■ **Example 2.3**

```
stack.Pop();
```

This is the stack on which pop() will be called.

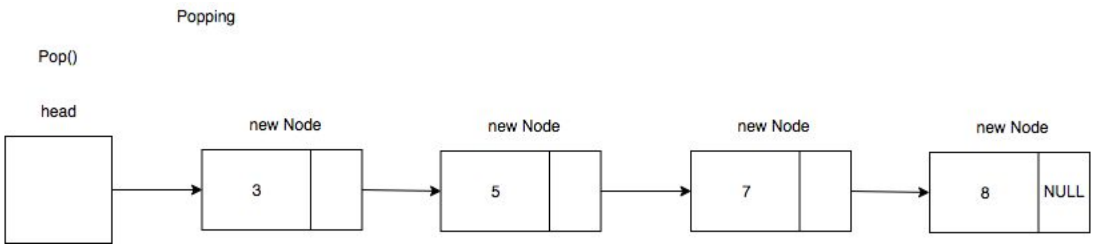


Figure 2.12: Stack before pop()

After checking the head pointer if it is not NULL, it creates a temp pointer and stores the address of node pointed by head in it.

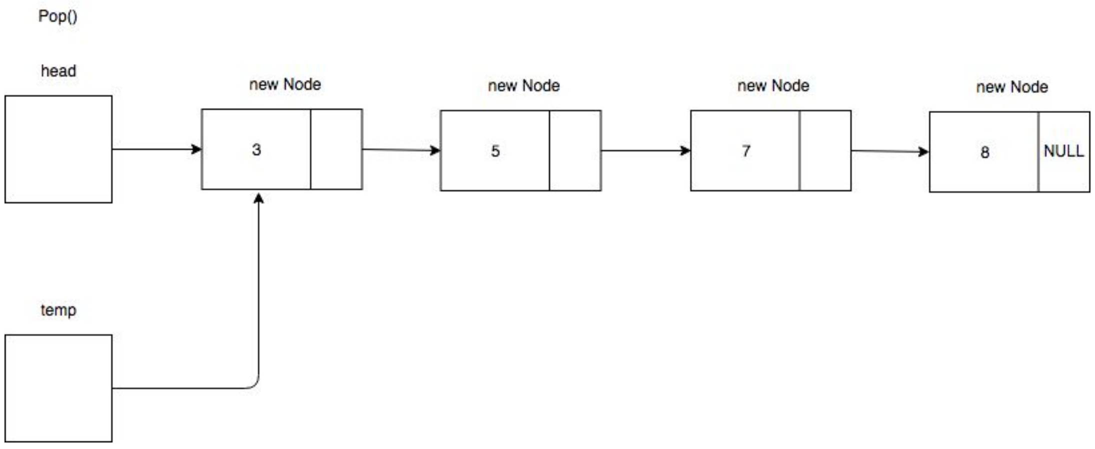


Figure 2.13: Creating temp pointer

It updates the head pointer to the node next to head

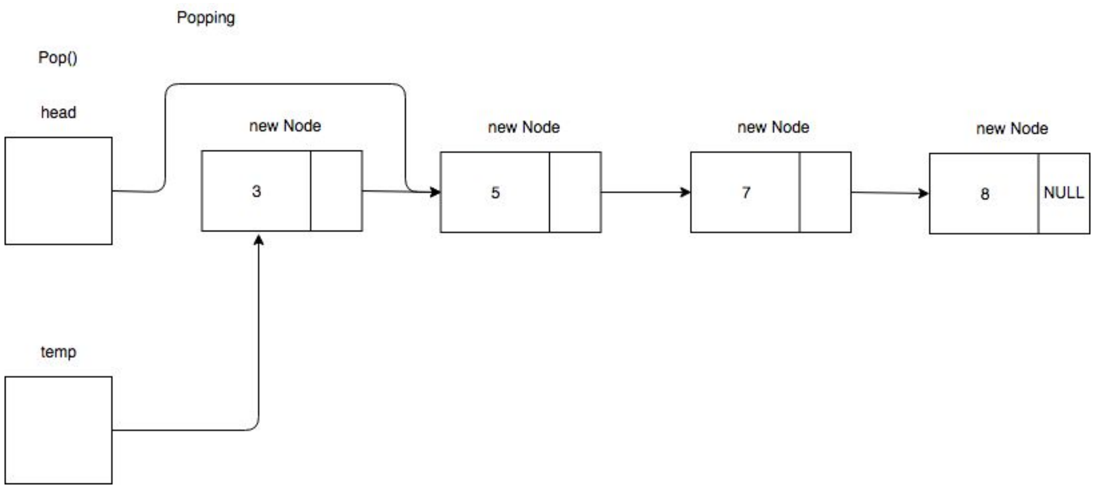


Figure 2.14: Updating head pointer

It now deletes the node pointed by temp.

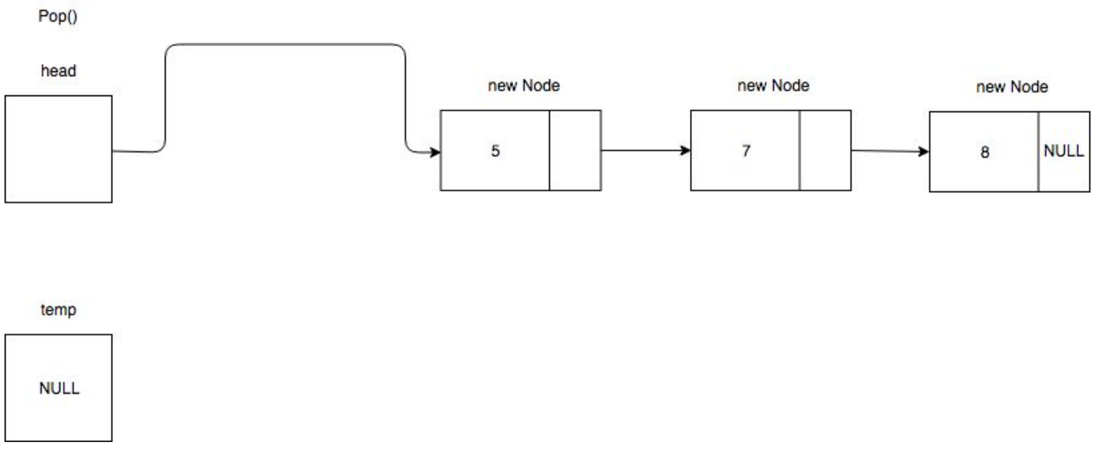


Figure 2.15: Deleting the node pointed by temp

The temp pointer then gets destroyed and stack is updated successfully.

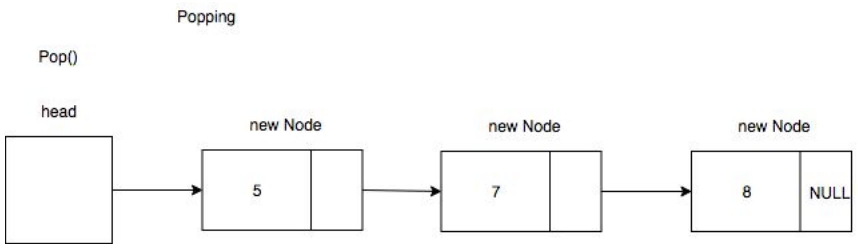


Figure 2.16: Succesful pop() operation

■

Implementing Show() function

The Show() function allows to output all the values in the stack.
Inside the class of Stack, we will declare and define a function "void Show()" in public.

```
void Show()
    Node* temp = head;
    while(temp!=NULL)
    {
        std::cout<<temp->data<<std::endl;
        temp = temp->next;
    }
```

Let's visualize an example of Show() operation

■ **Example 2.4**

```
    stack.Show();
```

Creating the temp pointer and points it to the node pointed by head.

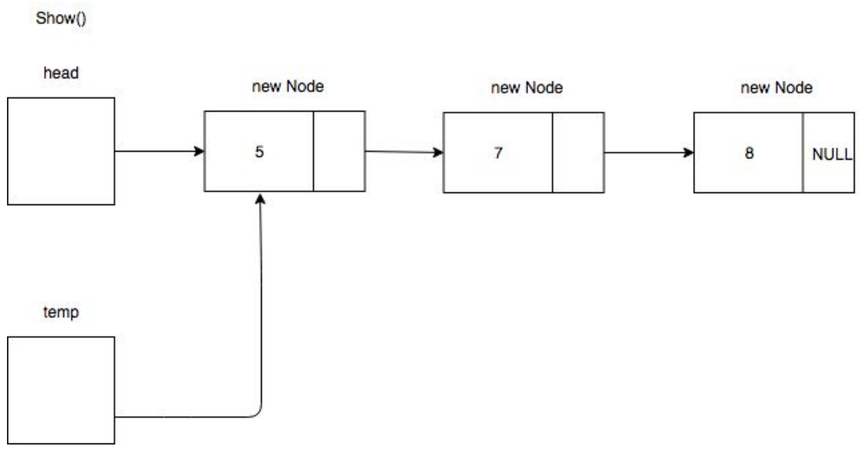


Figure 2.17: Creating temp pointer

Now it outputs the vaue stored in the node pointed by temp.

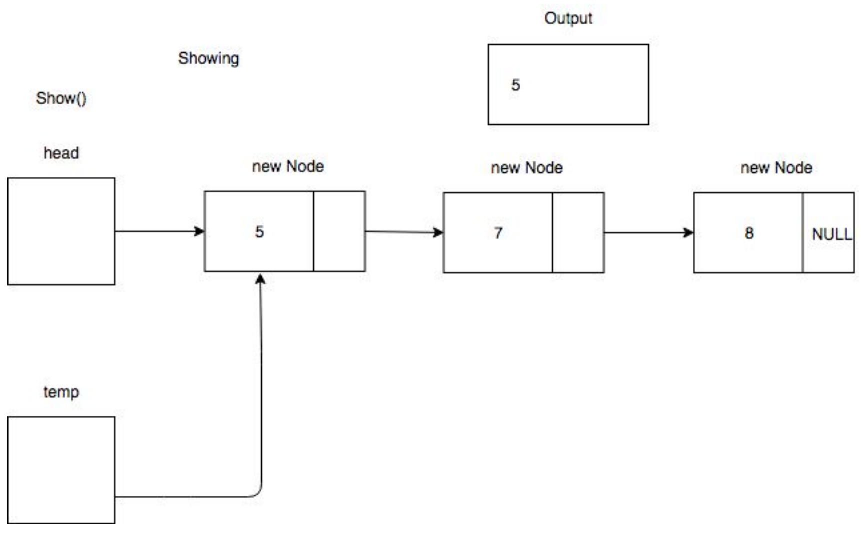


Figure 2.18: Outputting first value

It updates the temp pointer and moves to the next node. Then it outputs the value of the node pointed by temp.

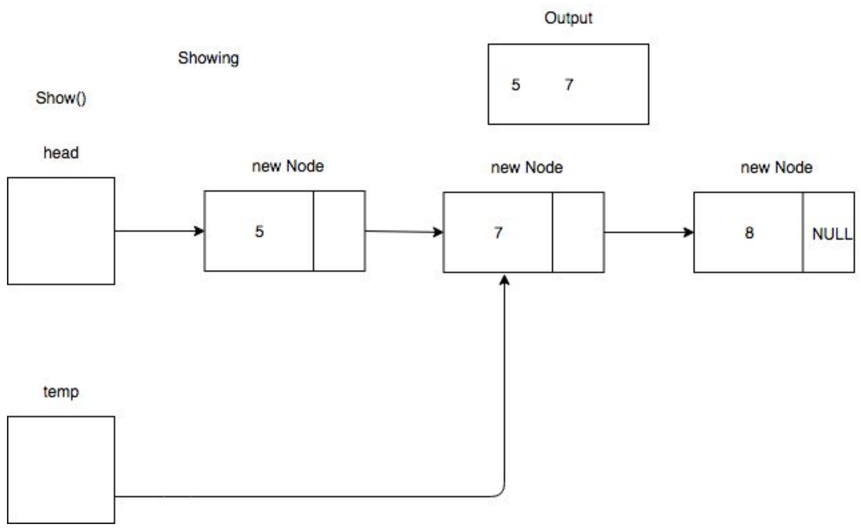


Figure 2.19: Updating temp pointer and outputting value

It repeats the updating of temp pointer and outputs the value

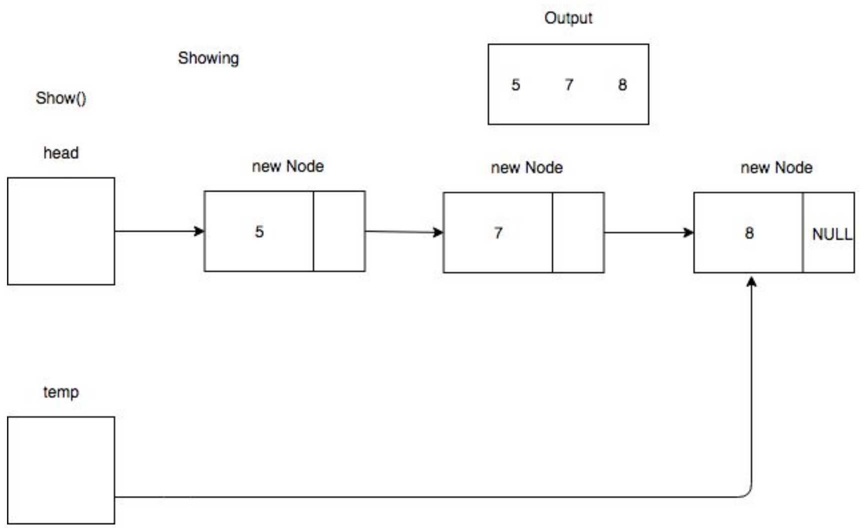


Figure 2.20: Updating the temp pointer to the third node and outputs the value

The loop gets stopped now because the temp is now set to NULL.

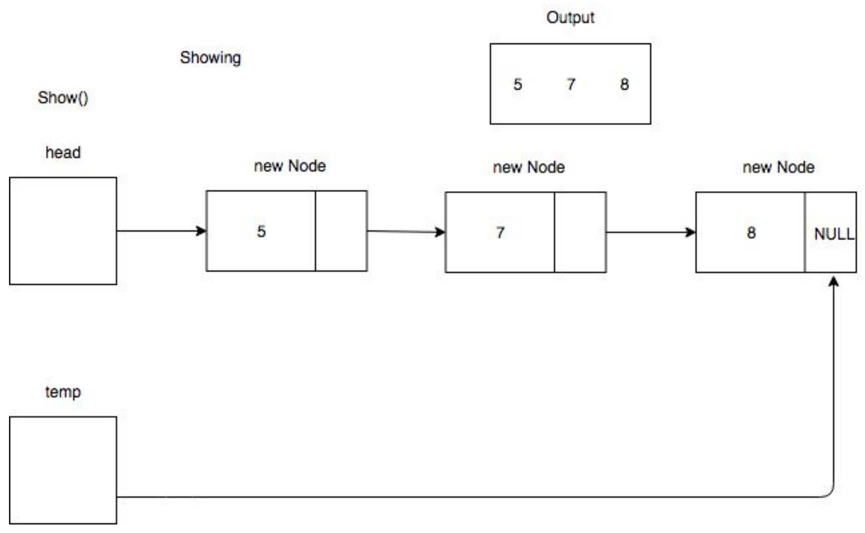


Figure 2.21: Updating the temp pointer to the third node and outputs the value

■

2.3 Problems

Problem 2.1 Implement the isEmpty() function in your Stack class. The function should return true if the stack is empty and return false if not.

Instructions

- Think about the value stored in head pointer when the stack is Empty

Problem 2.2 Implement the functions Top() and Bottom() in your Stack class. The Top() function should return the top element of the stack. The Bottom() function should return the element on the bottom of the stack.

Instructions

- Top() function is easy to implement after doing the problem above. Use head pointer to implement this function
- For Bottom() function, think how to iterate over the stack till the bottom using only head pointer and a temporary pointer.

2.4 Feedback

Please write the things you've learned from this lab and suggestions to make it more better and easy to learn.

3. Double-linked list

3.1 Objective

In this lab, we'll learn about a data structure, called Double linked list. We'll learn about its implementation, operations and uses.

3.2 Description

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence. It has the same operations as the single-linked list, but the opertions takes less time as there ire shorter traversals using links to both next and previous element. We'll see it in the examples below.

3.2.1 Real-life examples of Double linked list

There are many real life examples of stack. Consider the simple example of a music player. The music player allows skipping to both next and previous song in the playlist. Moreover, it allows to traverse through the list to find a song using both next and previous buttons.

3.2.2 Time Complexities of Double linked list

Append(data) - O(1) time
Add(i, data) - O(min(i, n - i))
Remove(i) - O(min(i, n - i))
Where i is the index and n is the size of the list.

For add and remove, we have to find the node at the index where a node is getting added or removed. Finding the node with a particular index in a DLList is easy; we can either start at the head of the list and work forward ,or start at the tail of the list and work backward. This allows us to reach the ith node in O(min(i,n - i)) time.

3.2.3 Applications of Double linked list

There are many possible applications of stack. Some are listed below:

- The browser cache which allows you to hit the BACK buttons (a linked list of URLs).
- Applications that have a Most Recently Used (MRU) list (a linked list of file names).
- A stack, hash table, and binary tree can be implemented using a doubly linked list.
- Undo functionality in Photoshop or Word (a linked list of state).

3.3 Implementation

We will be implementing a double linked list. In double linked list, we have nodes linked to each other. Each node stores a data and links to the node next to and previous to it.

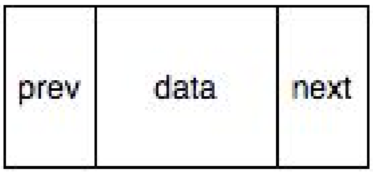


Figure 3.1: Representation of node

Implementing Node

To implement node, we will be making a struct to hold data and the links to store next and previous node . In this implementation we are making a node to store integer type data. However, it can be implemented to store any data type or object.

```
struct Node
{
    int data;
    Node* next;
    Node* prev;
    Node()
    {
        next = NULL;
        prev = NULL;
    }
};
```

Initializing the Double linked list

As a Double linked consists of many nodes, we should have many nodes inside our DLList class, but we will have only two nodes, "head" and "tail", which will act as reference nodes and allows us to iterate through all the nodes in the DLList. Moreover, we will include an integer "Size" to keep track of the size of the list.

```
class DLList
{
    private:
        int Size;
        Node* head;
        Node* tail;
    public:
        DLList()
        {
            head = NULL;
            tail = NULL:
        }
}
```

Our stack now consists of Node pointers which will hold the addresses of the head and the tail. To initialize, both are set to **NULL**, as the list is empty.

Figure 3.2: DLList head pointer

Figure 3.3: DLList tail pointer

Implementing Append operation

Inside the class of DLList, we will declare and define a function "Append(int data)" in public.

```
void Append(int data)
{
    if (Size == 0)
    {
        head = tail = new Node();
        head->data = data;
    }
    else
    {
        tail->next = new Node();
        tail->next->prev = tail;
        tail = tail->next;
        tail->data = data;
    }
    Size++;
}
```

■ **Example 3.1** Let's visualize an example Append() operation

```
    DLList lst;
    lst.Append(6);
```

Since the DLList is empty and head is set to NULL. Body of 'if' will be executed.
It first creates a new node and sets head and tail to it.

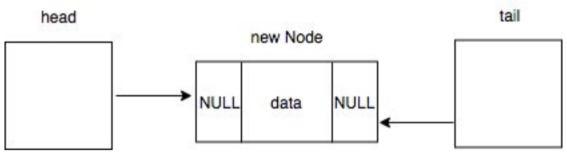


Figure 3.4: Creating new node

Then it sets its data to the data passed.

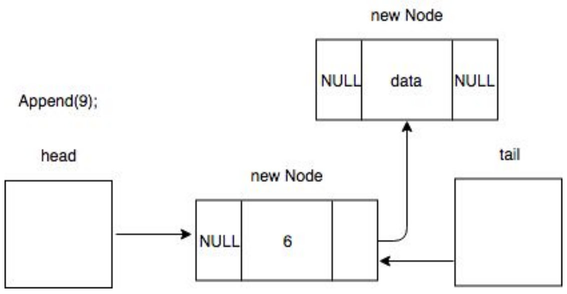


Figure 3.5: Setting data

■

Since, Size is not '0', Append operation will now execute body of 'else'. Let's see an example

■ **Example 3.2**

```
lst.append(9);
```

It will first create a new node and sets tail's next to it.

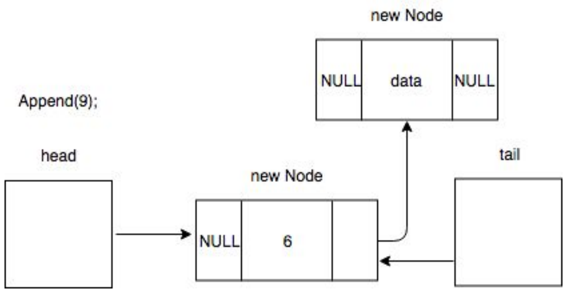


Figure 3.6: Creating new node

Then it sets the new node's prev to the tail

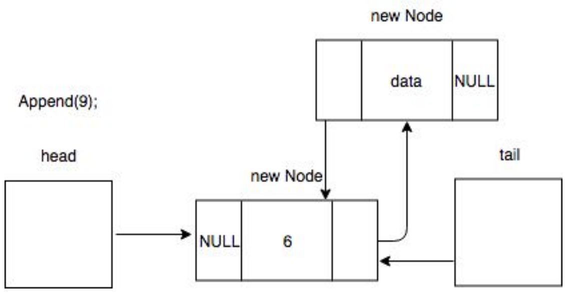


Figure 3.7: Setting new node's pointer

It now updates the tail and sets data.

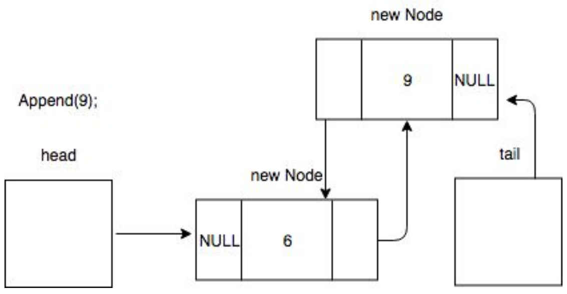


Figure 3.8: Updating tail and data

’9’ is added succesfully.

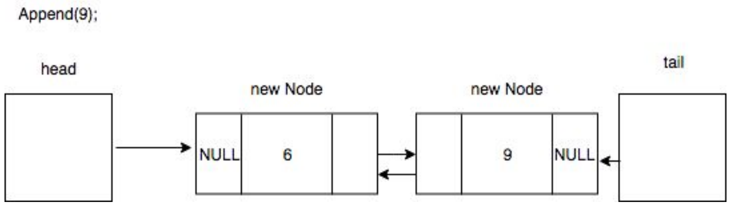


Figure 3.9: Succesful Append(9) operation

■

Implementing Add operation

Inside the class of DLList, we will declare and define a function "Append(int data)" in public.

```
void Add(int i, int data)
{
    if(i >= 0 && i <= Size)
    {
        if (i == 0)
        {
            if (Size == 0)
            {
                head = tail = new Node();
                head->data = data;
            }
            else
            {
                head->prev = new Node;
                head->prev->next = head;
                head = head->prev;
                head->data = data;
            }

        }
        else if (i == Size)
        {
            tail->next = new Node();
            tail->next->prev = tail;
            tail = tail->next;
            tail->data = data;
        }
        else
        {
            Node* temp;
            if (i <= Size/2)
            {
                temp = head;
                for (int j = 0; j < i; j++)
                {
                    temp=temp->next;
                }
            }
            else
            {
                temp = tail;
                for (int j = 0; j < Size-i-1; j++)
                {
                temp = temp->prev;
                }
            }

            Node* newNode = new Node;
            newNode->data = data;
            newNode->prev = temp->prev;
            newNode->next = temp;
            temp->prev->next = newNode;
            temp->prev = newNode;

        }
        Size++;
    }
    else
    {
        cout << "Index is out of range" << endl;
    }
}
```

■ **Example 3.3** Let's visualize an example Add() operation

```
    DLList lst;
    lst.Add(0,6);
```

Since the DLList is empty, Body of 'if (i == 0) and if (Size == 0)' will be executed.
It wll carry out the same operations carried out in Append function.
It first creates a new node and sets head and tail to it.

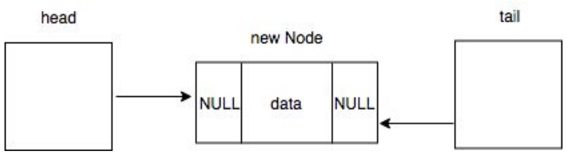


Figure 3.10: Creating new node

Then it sets its data to the data passed.

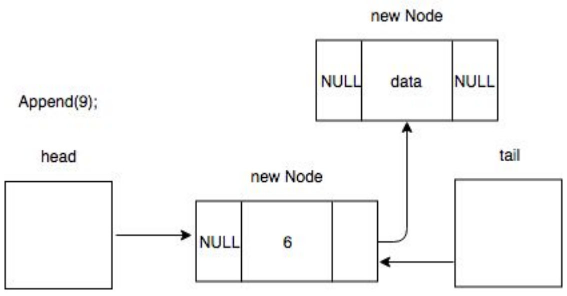


Figure 3.11: Setting data

■

If a number is added to the last, it will then also do the same tasks as append.

■ Example 3.4

```
lst.Add(1,9);
```

Now size is not 0 and index is equal to the size.
It will first create a new node and sets tail's next to it.

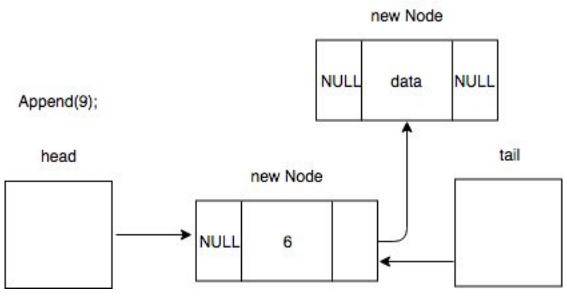


Figure 3.12: Creating new node

Then it sets the new node's prev to the tail

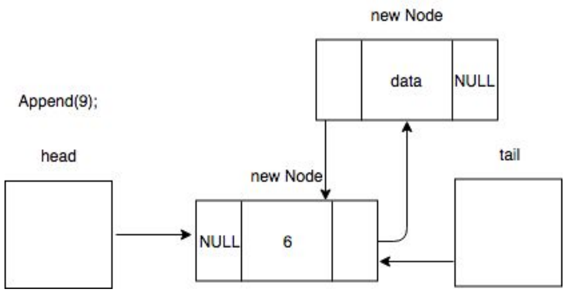


Figure 3.13: Setting new node's pointer

It now updates the tail and sets data.

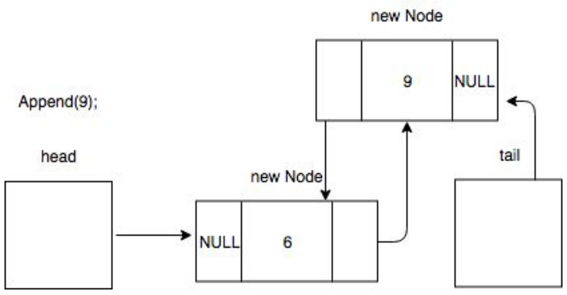


Figure 3.14: Updating tail and data

’9’ is added succesfully.

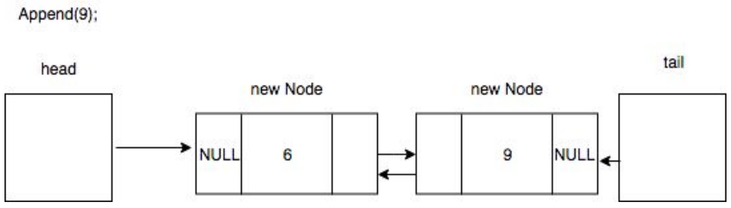


Figure 3.15: Succesful Add(1,9) operation

■

■ **Example 3.5** Let’s look at an example, when index is neither 0 nor equal to the size. Let’s visualize an example Add() operation

```
lst.Add(1,5);
```

Since i is less than half of the size, it will iterate from the head.
It will first create a temporary pointer and point it to head.

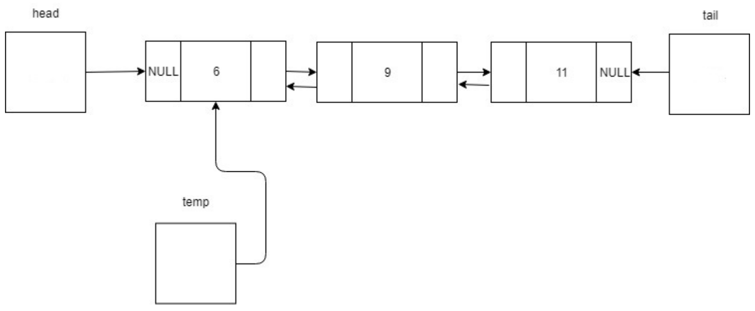


Figure 3.16: Creating a temporary pointer

It will iterate till it reaches the index where a new node is to be added.

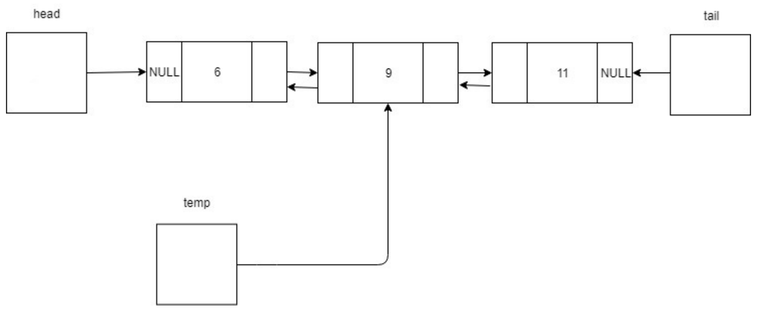


Figure 3.17: Iterating till it reaches the node

It will create a new Node.

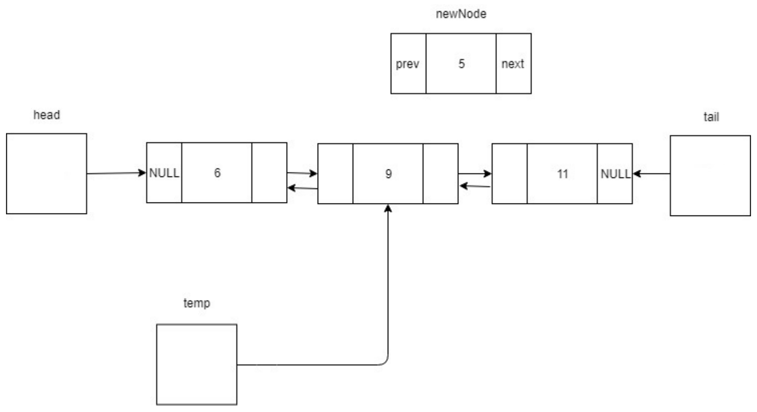


Figure 3.18: Creating new node

Points its ’prev’ pointer to the temp’s ’prev’.

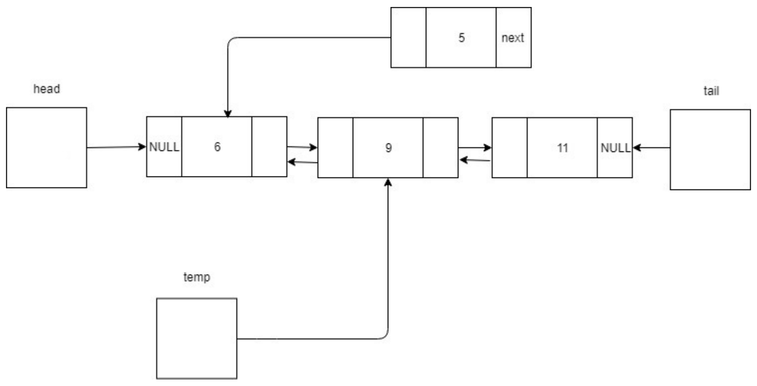


Figure 3.19: Setting node’s prev pointer

Points its ’next’ pointer to temp.

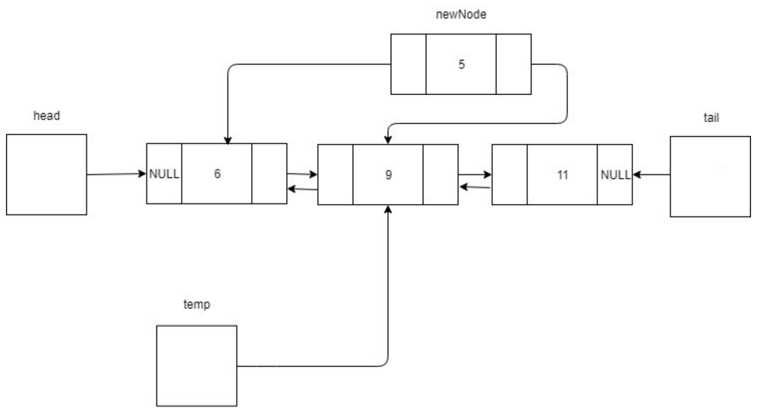


Figure 3.20: Setting node's next pointer

Point Node's prev's next pointer to itself.

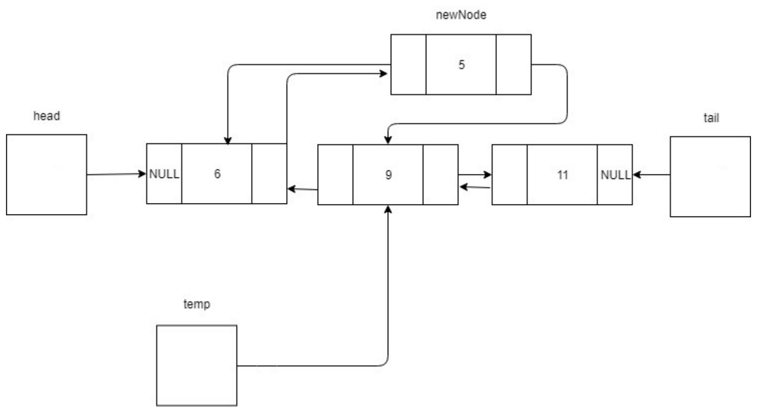


Figure 3.21: Setting the previous node's next pointer

And finally temp's prev to the new node.

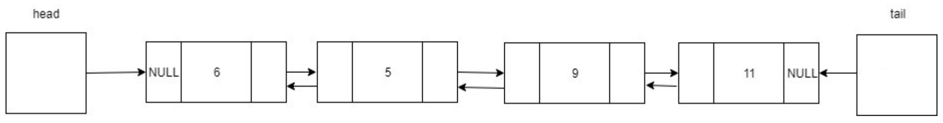


Figure 3.22: Succesful addition

■

■ **Example 3.6** Now, let's see an example where index is greater than the half of the size of the list.

```
lst.Add(2,13);
```

Since index is greater than the half of the size, it will start iteration from the last.
It will first create a temporary pointer and sets it to the tail.

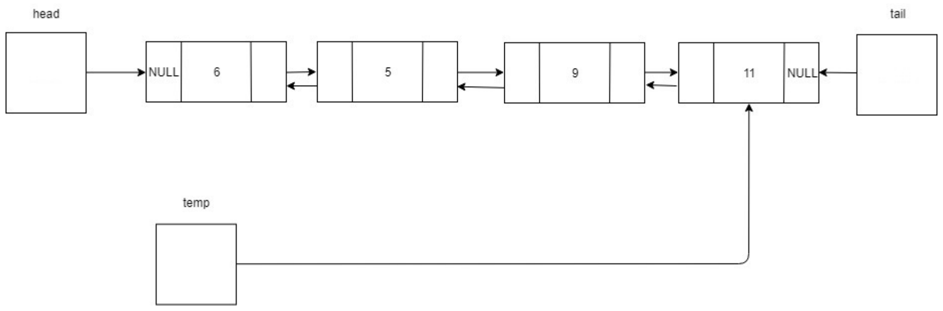


Figure 3.23: Creating a temporary pointer

It will iterate till it reaches the index where the new node has to be added

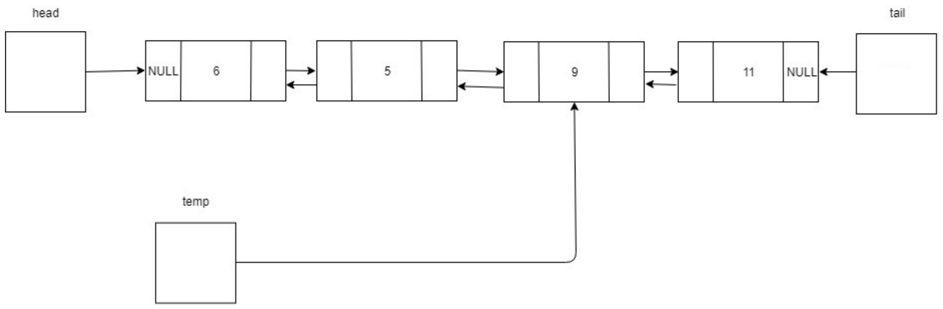


Figure 3.24: Iterated till the index

Now it repeats the same process of setting pointers as it did above and adds the new node succesfully.

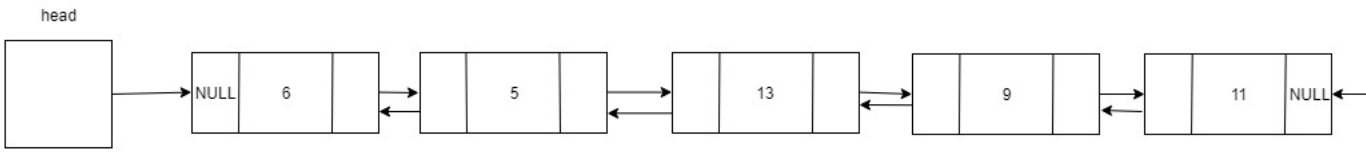


Figure 3.25: Succesful Add(2,13) operation

■

Let’s see an example when the size is not ’0’ and the node is to be added at index ’0’.

■ Example 3.7

```
lst.Add(0,4);
```

It will first create a new node and sets the prev’s pointer, of the node pointed by head, to it.

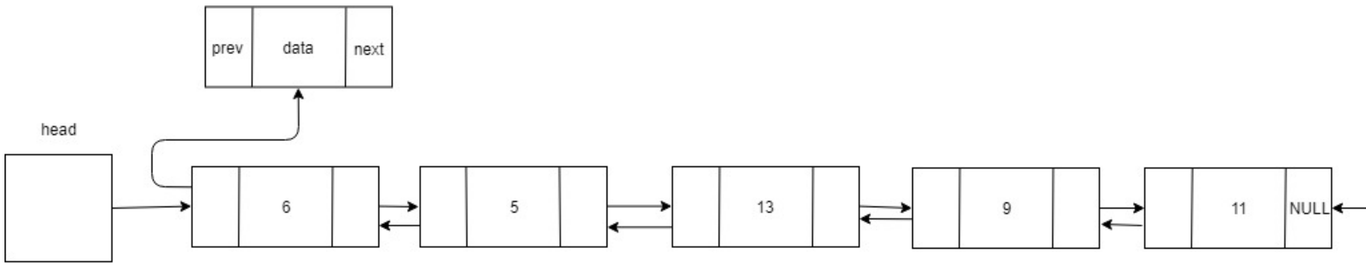


Figure 3.26: Creating new node and setting head's prev

Then it sets new node's next to the next of node pointed by head.

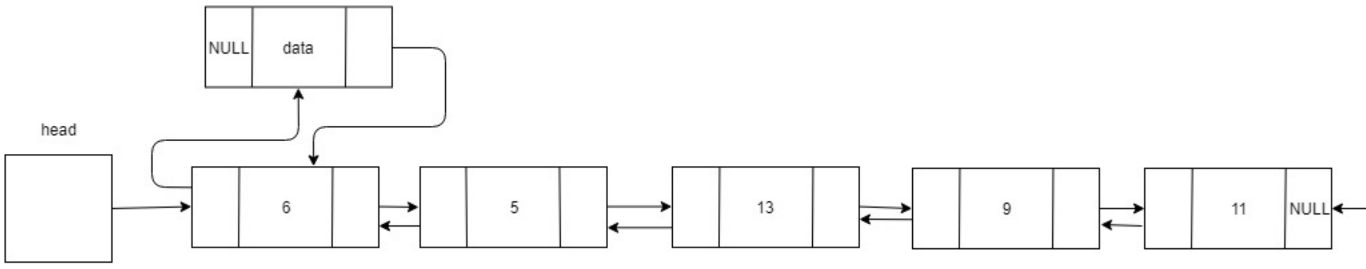


Figure 3.27: Setting node's next pointer

Finally, head is set to the new node and the node is added succesfully.

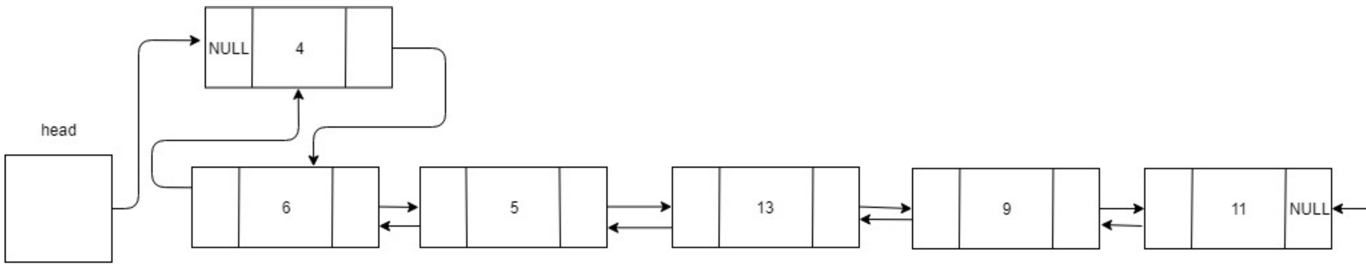


Figure 3.28: Updating head

■

Implementing Show() function

The Show() function allows to output all the values in the DLList.
Inside the class of DLList, we will declare and define a function "void Show()" in public.

```
void Show()
{
    Node* temp = head;
    while(temp!=NULL)
    {
        std::cout<<temp->data<<std::endl;
        temp = temp->next;
    }
}
```

Let’s visualize an example of Show() operation

■ **Example 3.8**

```
lst.Show();
```

It creates a temp pointer and is pointed to the node pointed by head. It also outputs its value

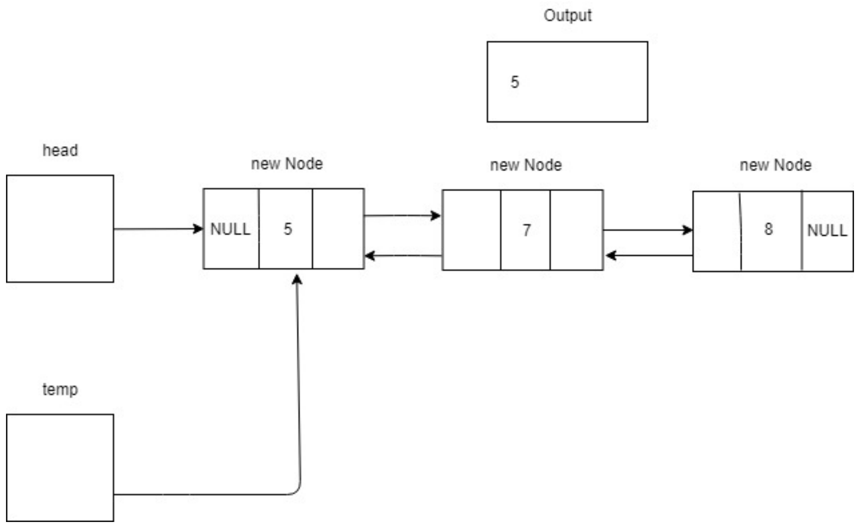


Figure 3.29: Creating temp pointer and outputting first value

It updates the temp pointer and moves to the next node. Then it outputs the value of the node pointed by temp.

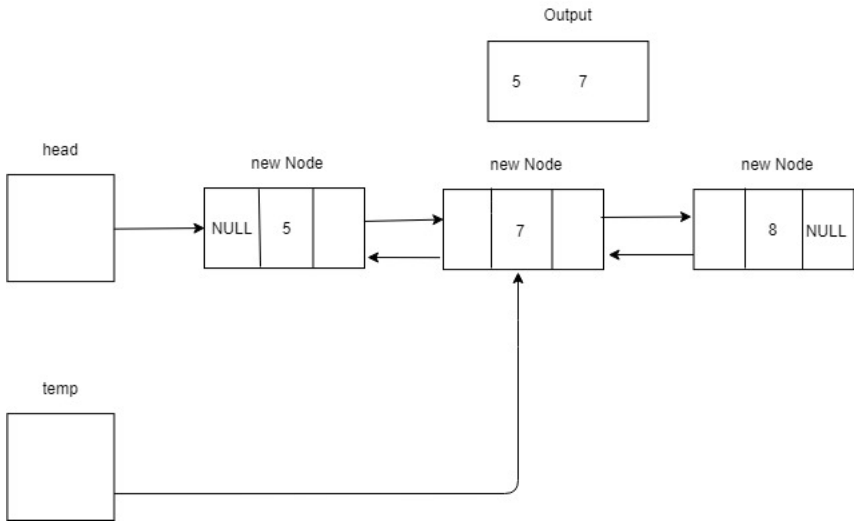


Figure 3.30: Outputting second value

And finally it moves to the last node, outputs its value and since it’s next is equal to NULL, the loop is ended.

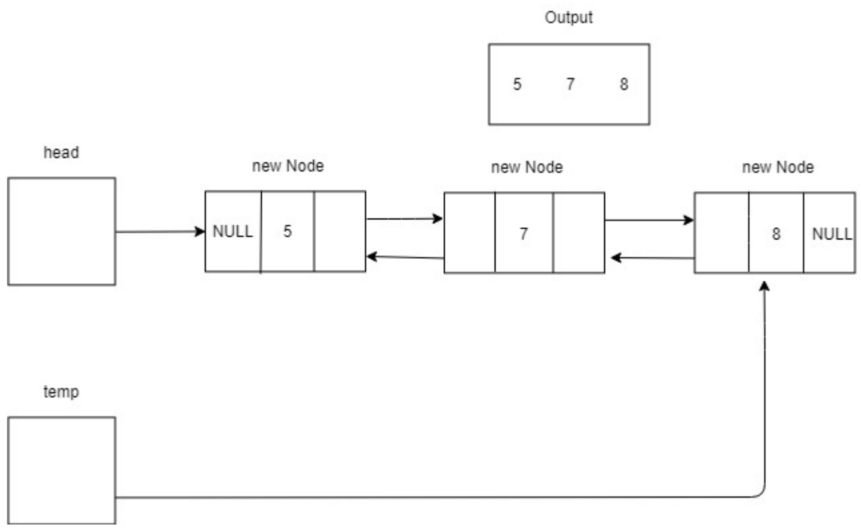


Figure 3.31: Outputting third value

■

3.4 Problems

Problem 3.1 Implement the Remove(int index) function in your DLList class. The function should remove the element from the given index and returns the removed element. The function should run in O(min(i, n - i)).

Instructions

- Think for all special cases such as, removal at zero index and removal at last index.
- Update pointers carefully.
- Try to take an idea from given Add() function for traversal.

Problem 3.2 Implement the function Reverse() which reverses the list, without making a new list.

Instructions

- Traverse through the whole list and update its pointers accordingly

3.5 Feedback

Please write the things you've learned from this lab and suggestions to make it more better and easy to learn.

4. Abstract classes

4.1 Objective

In this lab, we will learn about a type of class called abstract classes and its usage and implementation.

4.2 Description

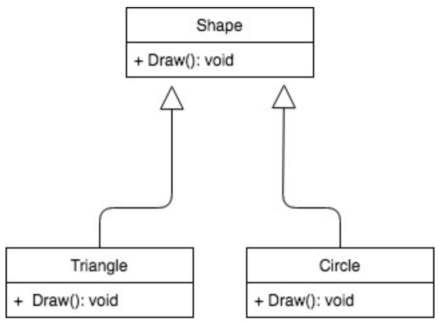


Figure 4.1: UML representation of abstract class

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called **abstract class**. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes. For this, we need to have at least one pure virtual function in the class.
A **pure virtual function** (or abstract function) in C++ is a virtual function for which we don't have

implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. It's done by the syntax showed below.

```
// An abstract class
class Test
{
    // Data members of class
    public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Let's see an example in whch a pure virtual function is declared in the abstract class and then defined in the derived class.

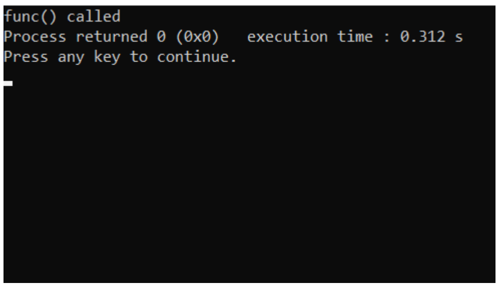
■ Example 4.1

```
class Base
{
    int x;
    public:
        virtual void func() = 0;
        int getX()
        {
            return x;
        }
};

// This class inherits from Base and implements func()
class Derived: public Base
{
    int y;
    public:
        void func()
        {
            cout << "func() called";
        }
};

int main()
{
    Derived d;
    d.func();
    return 0;
}
```

Now the abstract class has the attribute 'int x' and the function 'int GetX()'. This function will also be the part of derived class and is already defined in the base class. Now the function 'func()' is a pure virtual function which is defined in the base class and is implemented in the derived class. Now, if we create an instance of Derived class, we can access the functions 'GetX()' and 'func()'. The 'GetX()' will be called through the base class and 'func()' will be called through the derived class. The program above will output:



```
func() called
Process returned 0 (0x0)   execution time : 0.312 s
Press any key to continue.
```

Figure 4.2: Output of implemented pure virtual function

■

The derived class can also become an abstract class if the pure virtual function declared in base class is not defined in the derived class.

■ Example 4.2

```
class Base
{
    public:
    virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
    Derived d;
    return 0;
}
```

■

Corollary 4.2.1 — Interface vs Abstract classes. An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

4.3 Problems

Problem 4.1 Make an abstract class of Shape with pure virtual function to calculate area. It should also have the attributes for the position of the Shape. The derived classes should be Square, Circle, Trapezium and Rectangle. All derived classes should implement the pure virtual function defined in the base class.

Instructions

- Make a struct for Position to have x and y coordinates.
- Define the attributes for dimensions according to the shape in derived classes

Problem 4.2 Make a base class with a pure virtual function. Derive a class from it and declare another pure virtual function in the derived class. Then derive an other class from the first derived class and declare a pure virtual function in it too. And then lastly derive another class from the second derived class, and define all the pure virtual functions in it

4.4 Feedback

Please write the things you've learned from this lab and suggestions to make it more better and easy to learn.

5. In-text Elements

5.1 Theorems

This is an example of theorems.

5.1.1 Several equations

This is a theorem consisting of several equations.

Theorem 5.1.1 — Name of the theorem. In $E=\mathbb{R}^n$ all norms are equivalent. It has the properties:

$$\big|||\mathbf{x}||-||\mathbf{y}||\big| \leq ||\mathbf{x}-\mathbf{y}|| \tag{5.1}$$

$$||\sum_{i=1}^{n}\mathbf{x}_i|| \leq \sum_{i=1}^{n}||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \tag{5.2}$$

5.1.2 Single Line

This is a theorem consisting of just one line.

Theorem 5.1.2 A set $\mathscr{D}(G)$ in dense in $L^2(G)$, $|\cdot|_0$.

5.2 Definitions

This is an example of a definition. A definition could be mathematical or it could define a concept.

Definition 5.2.1 — Definition name. Given a vector space E, a norm on E is an application, denoted $||\cdot||$, E in $\mathbb{R}^+=[0,+\infty[$ such that:

$$||\mathbf{x}||=0 \Rightarrow \mathbf{x}=\mathbf{0} \tag{5.3}$$

$$||\lambda\mathbf{x}||=|\lambda|\cdot||\mathbf{x}|| \tag{5.4}$$

$$||\mathbf{x}+\mathbf{y}|| \leq ||\mathbf{x}||+||\mathbf{y}|| \tag{5.5}$$

5.3 Notations

Notation 5.1. *Given an open subset G of $\mathbb{R}^n$, the set of functions φ are:*

1. *Bounded support G;*
2. *Infinitely differentiable;*

a vector space is denoted by $\mathscr{D}(G)$.

5.4 Remarks

This is an example of a remark.

R The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field $\mathbb{K} = \mathbb{R}$, however, established properties are easily extended to $\mathbb{K} = \mathbb{C}$.

5.5 Corollaries

This is an example of a corollary.

Corollary 5.5.1 — Corollary name. The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field $\mathbb{K} = \mathbb{R}$, however, established properties are easily extended to $\mathbb{K} = \mathbb{C}$.

5.6 Propositions

This is an example of propositions.

5.6.1 Several equations

Proposition 5.6.1 — Proposition name. It has the properties:

$$\big| ||\mathbf{x}|| - ||\mathbf{y}|| \big| \leq ||\mathbf{x} - \mathbf{y}|| \tag{5.6}$$

$$|| \sum_{i=1}^{n} \mathbf{x}_i || \leq \sum_{i=1}^{n} ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \tag{5.7}$$

5.6.2 Single Line

Proposition 5.6.2 Let $f, g \in L^2(G)$; if $\forall \varphi \in \mathscr{D}(G)$, $(f, \varphi)_0 = (g, \varphi)_0$ then $f = g$.

5.7 Examples

This is an example of examples.

5.7.1 Equation and Text

■ **Example 5.1** Let $G = \{x \in \mathbb{R}^2 : |x| < 3\}$ and denoted by: $x^0 = (1,1)$; consider the function:

$$f(x) = \begin{cases} \mathrm{e}^{|x|} & \text{si } |x - x^0| \leq 1/2 \\ 0 & \text{si } |x - x^0| > 1/2 \end{cases} \tag{5.8}$$

The function f has bounded support, we can take $A = \{x \in \mathbb{R}^2 : |x - x^0| \leq 1/2 + \varepsilon\}$ for all $\varepsilon \in]0; 5/2 - \sqrt{2}[$. ■

5.7.2 Paragraph of Text

■ **Example 5.2 — Example name.** Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. ■

5.8 Exercises

This is an example of an exercise.

Exercise 5.1 This is a good place to ask a question to test learning progress or further cement ideas into students' minds. ■

5.9 Problems

Problem 5.1 What is the average airspeed velocity of an unladen swallow?

5.10 Vocabulary

Define a word to improve a students' vocabulary.
Vocabulary 5.1 — Word. Definition of word.

Part Two

6. Presenting Information

6.1 Table

Treatments	**Response 1**	**Response 2**
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

Table 6.1: Table caption

6.2 Figure

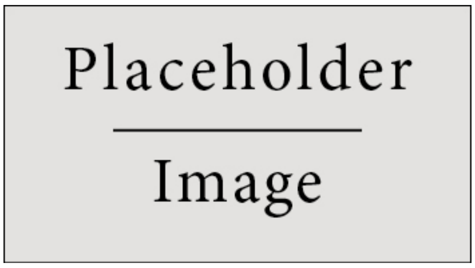


Figure 6.1: Figure caption

Bibliography

Index