# Problem Solving and Programming 2

## Assignment 2:

## Name: CHO, SEO YOON ID: 1909355

| | Element | Description | Mark | Your mark |
|---|---|---|---|---|
| **1** | Linked list + Stack & Queue | You need to develop the functionality for the linked list which involve, creating the list, and the main functionality, Insert, Append, Delete, DeleteAll, and ShowList. The you need to implement the functions for Stack and Queue which includes StackPush, stackPop, QueuePush, and QueuePop. | 20 | |
| **2** | Doubly Linked list + Stack & Queue | **Same as 1, but using Doubly linked list** | 20 | |
| **3** | Binary tree | You need to develop the functionality for the Binary Tree which involve, creating the BT, and the main functionality, Insert, Find, Delete, DeleteAll, GetNodeInfo, and ShowTree in an inorder traverse. | 30 | |
| **4** | Presentation of the code | This will be concerning the way you have presented your code and how it is structured | 10 | |
| **5** | Comments used | This will be concerning the comments you are adding to describe each function and your comments on the critical parts within your functions. Do not over comment your code | 10 | |
| **6** | Screen shoots for the output | This will show that your programs are running and having the desired output. | 10 | |
| | | **Total** | **100** | |

The assignment should be submitted as a word document includes your code for the above and a screenshot for your output for each program you had in your submissions.

# Problem Solving and Programming 2

## Single List

```
#include <iostream>
#include <stdio.h>
#include <string>
using namespace std;

//Structure declaration
struct ListNode
{
	float value;
	struct ListNode* next;
};

//Pointer initialisation
ListNode* head = NULL;

//Functions
ListNode* createNode(ListNode* head);
ListNode* insertNode(ListNode* head, float num);
ListNode* appendNode(ListNode* head, float num);
ListNode* deleteNode(ListNode* head, float num);
ListNode* deleteList(ListNode* head);
void	  displayList(ListNode* head);

int main()
{
	float num;
	int selection(0);

	cout << "Node Creation : -------------------------\n";
	head = createNode(head);
	cout << "Created Node : \n";
	displayList(head);

	do
	{
		cout << "\n\nSelect One : -------------------------\n";
		cout << "1. Ascending Insert\n";
		cout << "2. Append\n";
		cout << "3. Delete\n";
		cout << "4. Delete All\n";
		cout << "5. Display\n";
		cout << "6. Exit\n";
		cin >> selection;

		switch (selection)
		{
		case 1:
			cout << "\nNode Insertion : ----------------------\n";
			cout << "Input Number to Insert : ";
			cin >> num;
			head = insertNode(head, num);
			cout << "END";
			break;
		case 2:
			cout << "\nNode Append : -------------------------\n";
			cout << "Input Number to Append : ";
			cin >> num;
			head = appendNode(head, num);
			cout << "END";
			break;
```

```
		case 3:
			cout << "\nNode Deletion : ----------------------\n";
			cout << "Input Node Value to Delete : ";
			cin >> num;
			head = deleteNode(head, num);
			cout << "END";
			break;
		case 4:
			cout << "\nList Deletion : ----------------------\n";
			head = deleteList(head);
			cout << "END";
			break;
		case 5:
			cout << "\nList Display : ----------------------\n";
			displayList(head);
			cout << "END";
			break;
		default:
			break;
		}

	} while (selection != 6);

	cout << "\n\nEND OF THE PROGRAM\n\n";

}

ListNode* createNode(ListNode* head)
{
	float num;
	char ch;
	cout << "Insert Values (press Enter to finish) : ";
	do
	{
		cin >> num;
		//Input values create a node in ascending order
		head = insertNode(head, num);
		ch = getchar();
	} while (ch != '\n');

	return head;
}
ListNode* insertNode(ListNode* head, float num)
{
	ListNode* newNode, * nodePtr, * previousNode = NULL;

	//Allocate a new node
	//Store num
	newNode = new ListNode;
	newNode->value = num;
	newNode->next = NULL;

	if (head == NULL)   //If no node, make newNode the first node
		head = newNode;
	else
	{
		nodePtr = head;

		//Find num's place by skipping smaller values than num
		while (nodePtr != NULL && nodePtr->value < num)
		{
			previousNode = nodePtr;
```

```
                        nodePtr = nodePtr->next;
                }
                //If the newNode = smallest,
                //Let it be the first node
                if (previousNode == NULL)
                {
                        head = newNode;
                        newNode->next = nodePtr;
                }
                else   //If not first, just insert it
                {
                        previousNode->next = newNode;
                        newNode->next = nodePtr;
                }
        }
        return head;
}
ListNode* appendNode(ListNode* head, float num)
{
        ListNode* newNode, * nodePtr;

        //Allocate a new node
        //Store num
        newNode = new ListNode;
        newNode->value = num;
        newNode->next = NULL;

        if (head == NULL)   //If no node, make newNode the first node
                head = newNode;
        else
        {
                nodePtr = head;
                // Find the last node in the list
                while (nodePtr->next != NULL)
                        nodePtr = nodePtr->next;
                // Insert newNode as the last node
                nodePtr->next = newNode;
        }
        return head;
}
ListNode* deleteNode(ListNode* head, float num)
{
        ListNode* nodePtr, * previousNode = NULL;

        // If the list is empty, do nothing.
        if (head == NULL)
                return head;

        // If the firstNode is num
        if (head->value == num)
        {
                nodePtr = head->next;
                delete head;
                head = nodePtr;
        }
        else
        {
                nodePtr = head;
                //Find num's place by skipping smaller values than num
                while (nodePtr != NULL && nodePtr->value != num)
                {
                        previousNode = nodePtr;
```

```
                        nodePtr = nodePtr->next;
                }
                // Link the previousNode and the node after nodePtr
                // then delete nodePtr
                if (nodePtr != NULL)
                {
                        previousNode->next = nodePtr->next;
                        delete nodePtr;
                }
        }
        return head;
}
ListNode* deleteList(ListNode* head)
{
        ListNode* nodePtr, * nextNode;

        //Let nodePtr be the first node i.e. head
        nodePtr = head;

        //Delete nodePtr
        //then truncate it by cascading the function
        //itself to next node
        while (nodePtr != NULL)
        {
                nextNode = nodePtr->next;
                delete nodePtr;
                nodePtr = nextNode;
        }
        head = nodePtr;
        return head;
}
void displayList(ListNode* head)
{
        ListNode* nodePtr = head;
        //Recursive function to recall every node's value
        //by cascading
        while (nodePtr)
        {
                cout << nodePtr->value << endl;
                nodePtr = nodePtr->next;
        }
}
```

```
Node Creation : ----------------------------
Insert Values in Ascending Order (press En
Created Node :
1.1
3.3
5.5
7.7
9.9


Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
1

Node Insertion : ------------------------
Input Number to Insert : 2.2
END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : ----------------------
1.1
2.2
3.3
5.5
7.7
9.9
END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
2
```

```
Node Append : ----------------------------
Input Number to Append : 10.0
END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : ------------------------
1.1
2.2
3.3
5.5
7.7
9.9
10
END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
3

Node Deletion : ------------------------
Input Node Value to Delete : 2.2
END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5
```

# Problem Solving and Programming 2

```
List Display : ----------------------
1.1
3.3
5.5
7.7
9.9
10
END

Select One : --------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
4

List Deletion : ----------------------
END

Select One : --------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : ----------------------
END

Select One : --------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
6

END OF THE PROGRAM


C:₩Users₩yooni₩source₩repos₩Assignment2_Q1₩Debug₩Assignment2_Q1.exe (proc
To automatically close the console when debugging stops, enable Tools->Op
le when debugging stops.
Press any key to close this window . . .
```

# Problem Solving and Programming 2

## Single List – Stack and Que

```
#include <iostream>
#include <stdio.h>
#include <string>
using namespace std;

//Structure declaration
struct StackNode
{
	float value;
	struct StackNode* next;
};
struct QueueNode
{
	float value;
	struct QueueNode* next;
};
//Pointer initialisation
StackNode* SHead = NULL;
QueueNode* QHead = NULL;

//Functions
StackNode* StackPush(StackNode* head, float num);
StackNode* StackPop(StackNode* head);
void displayStack(StackNode* head);
QueueNode* QueuePush(QueueNode* head, float num);
QueueNode* QueuePop(QueueNode* head);
void displayQueue(QueueNode* head);

int main()
{
	float num;
	char ch;
	int selection(0);

	do
	{
		cout << "\n1. Stack Push\n";
		cout << "2. Stack Pop\n";
		cout << "3. Queue Push\n";
		cout << "4. Queue Pop\n";
		cout << "5. Exit\n";
		cin >> selection;
		switch (selection)
		{
		case 1:
			cout << "\nStack Push ------------------\n";
			cout << "Insert Values (press Enter to finish) : ";
			do
			{
				cin >> num;
				SHead = StackPush(SHead, num);
				ch = getchar();
			} while (ch != '\n');
			displayStack(SHead);
			break;
		case 2:
			cout << "\nStack Pop ------------------\n";
			SHead = StackPop(SHead);
			displayStack(SHead);
			break;
```

```
		case 3:
			cout << "\nQueue Push ------------------\n";
			cout << "Insert Values (press Enter to finish) : ";
			do
			{
				cin >> num;
				QHead = QueuePush(QHead, num);
				ch = getchar();
			} while (ch != '\n');
			displayQueue(QHead);
			break;
		case 4:
			cout << "\nQueue Pop ------------------\n";
			QHead = QueuePop(QHead);
			displayQueue(QHead);
			break;
		default:
			break;
		}
	} while (selection != 5);

	cout << "\n\nEND OF PROGRAM\n\n";
}

StackNode* StackPush(StackNode* head, float num)
{
	StackNode* newNode, * nodePtr;

	//Allocate a new stack
	//Store num
	newNode = new StackNode;
	newNode->value = num;
	newNode->next = NULL;

	if (head == NULL)   //If no node, make newNode the first stack
		head = newNode;
	else //otherwise make num go to the end
	{
		nodePtr = head;
		// Find the last node and then insert
		while (!nodePtr->next != NULL)
			nodePtr = nodePtr->next;
		nodePtr->next = newNode;
	}
	return head;
}
StackNode* StackPop(StackNode* head)
{
	StackNode* nodePtr, * previousNode = NULL;

	if (head == NULL)    // If the list is empty, do nothing.
		return head;
	else // Otherwise pop the last input value
	{
		nodePtr = head;
		while (nodePtr->next != NULL)
		{
			previousNode = nodePtr;
			nodePtr = nodePtr->next;
		}
		previousNode->next = NULL;
		delete nodePtr;
```

```
	}
	return head;
}
void displayStack(StackNode* head)
{
	StackNode* nodePtr;
	nodePtr = head;
	//Recursive function to recall every node's value
	//by cascading
	while (nodePtr)
	{
		cout << nodePtr->value << endl;
		nodePtr = nodePtr->next;
	}
}

QueueNode* QueuePush(QueueNode* head, float num)
{
	QueueNode* newNode, * nodePtr;

	//Allocate a new stack
	//Store num
	newNode = new QueueNode;
	newNode->value = num;
	newNode->next = NULL;
	if (head == NULL)   //If no node, make newNode the first stack
		head = newNode;
	else   //otherwise make num go to the end
	{
		nodePtr = head;
		// Find the last node and then insert
		while (nodePtr->next != NULL)
			nodePtr = nodePtr->next;
		nodePtr->next = newNode;
	}
	return head;
}
QueueNode* QueuePop(QueueNode* head)
{
	QueueNode* nodePtr;
	if (head == NULL)    // If the list is empty, do nothing.
		return head;
	else   // Otherwise pop the first input value
	{
		nodePtr = head->next;
		delete head;
		head = nodePtr;
		return head;
	}
}
void displayQueue(QueueNode* head)
{
	QueueNode* nodePtr;
	nodePtr = head;
	//Recursive function to recall every node's value
	//by cascading
	while (nodePtr)
	{
		cout << nodePtr->value << endl;
		nodePtr = nodePtr->next;
	}
}
```

# Problem Solving and Programming 2

```
1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
1

Stack Push --------------------
Insert Values (press Enter to finish) : 1 3 5 7 9 10
1
3
5
7
9
10

1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
2

Stack Pop --------------------
1
3
5
7
9

1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
3

Queue Push --------------------
Insert Values (press Enter to finish) : 2 4 6 8 9
2
4
6
8
9

1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
4

Queue Pop --------------------
4
6
8
9

1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
5

END OF PROGRAM

C:₩Users₩yooni₩source₩repos₩Assignment2_Q1
To automatically close the console when de
Press any key to close this window . . .
```

# Problem Solving and Programming 2

## Double List

```
#include <iostream>
#include <stdio.h>
#include <string>
using namespace std;

//Structure declaration
struct DListNode
{
	float value;
	struct DListNode* next;
	struct DListNode* prev;
};

//Pointer initialisation
DListNode* head = NULL;

//Functions
DListNode* createNode(DListNode* head);
DListNode* appendNode(DListNode* head, float num);
DListNode* insertNode(DListNode* head, float num);
DListNode* deleteNode(DListNode* head, float num);
DListNode* deleteList(DListNode* head);
void	displayList(DListNode* head);

int main()
{
	float num;
	int selection(0);

	cout << "Node Creation : -------------------------\n";
	head = createNode(head);
	cout << "Created Node : \n";
	displayList(head);

	do
	{
		cout << "\n\nSelect One : ------------------------\n";
		cout << "1. Ascending Insert\n";
		cout << "2. Append\n";
		cout << "3. Delete\n";
		cout << "4. Delete All\n";
		cout << "5. Display\n";
		cout << "6. Exit\n";
		cin >> selection;

		switch (selection)
		{
		case 1:
			cout << "\nNode Insertion : ----------------------\n";
			cout << "Input Number to Insert : ";
			cin >> num;
			head = insertNode(head, num);
			cout << "END";
			break;
		case 2:
			cout << "\nNode Append : -------------------------\n";
			cout << "Input Number to Append : ";
			cin >> num;
			head = appendNode(head, num);
			cout << "END";
```

```
                        break;
                case 3:
                        cout << "\nNode Deletion : ----------------------\n";
                        cout << "Input Node Value to Delete : ";
                        cin >> num;
                        head = deleteNode(head, num);
                        cout << "END";
                        break;
                case 4:
                        cout << "\nList Deletion : ----------------------\n";
                        head = deleteList(head);
                        cout << "END";
                        break;
                case 5:
                        cout << "\nList Display : ----------------------\n";
                        displayList(head);
                        cout << "END";
                        break;
                default:
                        break;
                }

        } while (selection != 6);

        cout << "\n\nEND OF THE PROGRAM\n\n";
}

DListNode* createNode(DListNode* head)
{
        float num;
        char ch;
        cout << "Insert Values (press Enter to finish) : ";
        do
        {
                cin >> num;
                //Input values create a node in ascending order
                head = insertNode(head, num);
                ch = getchar();
        } while (ch != '\n');
        return head;
}
DListNode* appendNode(DListNode* head, float num)
{
        DListNode* newNode, * nodePtr;

        //Allocate a new node
        //Store num
        newNode = new DListNode;
        newNode->value = num;
        newNode->next = NULL;
        newNode->prev = NULL;

        //If no node, make newNode the first node
        if (head == NULL)
                head = newNode;
        else
        {
                nodePtr = head;
                // Find the last node
                while (nodePtr->next != NULL)
                        nodePtr = nodePtr->next;
                // Insert newNode to the last
```

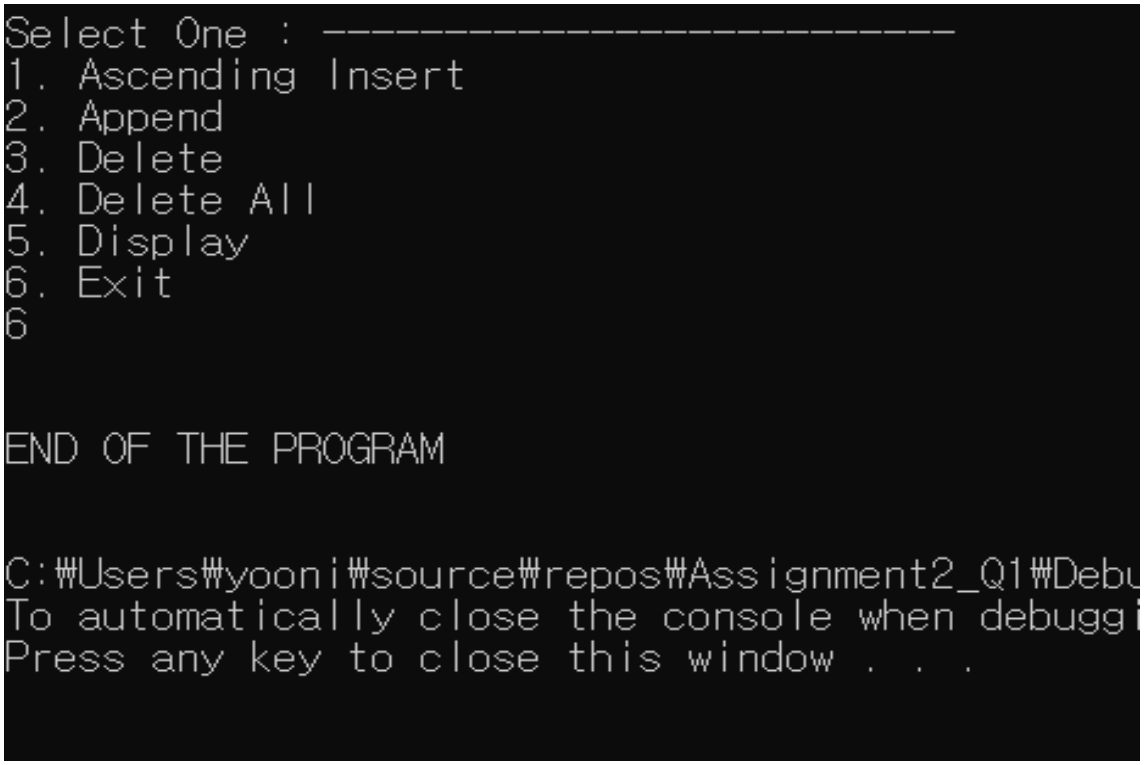
```
                nodePtr->next = newNode;
                newNode->prev = nodePtr;
        }
        return head;
}
void displayList(DListNode* head)
{
        DListNode* Tptr = head;

        //Recursive function to recall every node's value
        //by cascading
        while (Tptr != NULL)
        {
                cout << Tptr->value << endl;
                Tptr = Tptr->next;
        }
        cout << endl;
}
DListNode* insertNode(DListNode* head, float num)
{
        DListNode* newNode, * nodePtr;

        //Allocate a new node
        //Store num
        newNode = new DListNode;
        newNode->value = num;
        newNode->next = NULL;  newNode->prev = NULL;

        if (head == NULL)   //If no node, make newNode the first node
                head = newNode;
        else
        {
                nodePtr = head;

                //Find num's place by skipping smaller values than num
                while (nodePtr->next != NULL && nodePtr->value < num)
                        nodePtr = nodePtr->next;

                //If the newNode = smallest,
                //Let it be the first node
                if (nodePtr == head && nodePtr->value > num)
                {
                        nodePtr->prev = newNode;
                        newNode->next = nodePtr;
                        head = newNode;
                }
                else
                        // if the new node to be inserted at the end
                        if (nodePtr->next == NULL && nodePtr->value < num)
                        {
                                newNode->prev = nodePtr;
                                nodePtr->next = newNode;
                        }
                    // insert in the middle
                        else
                        {
                                newNode->next = nodePtr;
                                newNode->prev = nodePtr->prev;
                                nodePtr->prev->next = newNode;
                                nodePtr->prev = newNode;
                        }
        }
```

```
	return head;
}
DListNode* deleteNode(DListNode* head, float num)
{
	DListNode* nodePtr = head;

	// If the list is empty, do nothing.
	if (head == NULL)
		return head;

	// If the firstNode is num
	if (head->value == num)
	{
		head = nodePtr->next;
		if (head != NULL) head->prev = NULL;
		delete nodePtr;
		return head;
	}
	else
	{
		//Find num's place by skipping smaller values than num
		while (nodePtr->next != NULL && nodePtr->value != num)
			nodePtr = nodePtr->next;
		// Link the previousNode and the node after nodePtr
		// then delete nodePtr
		if (nodePtr->value == num)
		{
			nodePtr->prev->next = nodePtr->next;
			if (nodePtr->next != NULL)
				nodePtr->next->prev = nodePtr->prev;
			delete nodePtr;
		}
	}
	return head;
}
DListNode* deleteList(DListNode* head)
{
	DListNode* nodePtr, * nextNode;

	//Let nodePtr be the first node i.e. head
	nodePtr = head;

	//Delete nodePtr
	//then truncate it by cascading the function
	//itself to next node
	while (nodePtr != NULL)
	{
		nextNode = nodePtr->next;
		delete nodePtr;
		nodePtr = nextNode;
	}
	head = nodePtr;
	return head;
}
```

```
Node Creation : ---------------------------
Insert Values (press Enter to finish) : 3 5 7 1 2 6 9
Created Node :
1
2
3
5
6
7
9



Select One : ---------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
1

Node Insertion : ----------------------
Input Number to Insert : 4
END

Select One : ---------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : ----------------------
1
2
3
4
5
6
7
9

END
```

```
Select One : ---------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
2

Node Append : ---------------------------
Input Number to Append : 10
END

Select One : ---------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : ------------------------
1
2
3
4
5
6
7
9
10

END

Select One : ---------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
3

Node Deletion : ------------------------
Input Node Value to Delete : 4
END
```

```
Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : -------------------------
1
2
3
5
6
7
9
10

END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
4

List Deletion : ------------------------
END

Select One : ----------------------------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
5

List Display : -------------------------
List is empty...

END
```


Select One : ----------
1. Ascending Insert
2. Append
3. Delete
4. Delete All
5. Display
6. Exit
6
END OF THE PROGRAM
C:₩Users₩yooni₩source₩repos₩Assignment2_Q1₩Debu
To automatically close the console when debuggi
Press any key to close this window . . .

# Problem Solving and Programming 2

## Double List – Stack and Que

```
#include <iostream>
#include <stdio.h>
#include <string>
using namespace std;

//Structure declaration
struct StackNode
{
	float value;
	struct StackNode* next;
	struct StackNode* prev;
};
struct QueueNode
{
	float value;
	struct QueueNode* next;
	struct QueueNode* prev;
};

//Pointer initialisation
StackNode* SHead = NULL;
QueueNode* QHead = NULL;

//Functions
StackNode* StackPush(StackNode* head, float num);
StackNode* StackPop(StackNode* head);
void displayStack(StackNode* head);
QueueNode* QueuePush(QueueNode* head, float num);
QueueNode* QueuePop(QueueNode* head);
void displayQueue(QueueNode* head);

int main()
{
	float num;
	char ch;
	int selection(0);

	do
	{
		cout << "\n1. Stack Push\n";
		cout << "2. Stack Pop\n";
		cout << "3. Queue Push\n";
		cout << "4. Queue Pop\n";
		cout << "5. Exit\n";
		cin >> selection;
		switch (selection)
		{
		case 1:
			cout << "\nStack Push -------------------\n";
			cout << "Insert Values (press Enter to finish) : ";
			do
			{
				cin >> num;
				SHead = StackPush(SHead, num);
				ch = getchar();
			} while (ch != '\n');
			displayStack(SHead);
			break;
		case 2:
			cout << "\nStack Pop -------------------\n";
```

```
                SHead = StackPop(SHead);
                displayStack(SHead);
                break;
        case 3:
                cout << "\nQueue Push -------------------\n";
                cout << "Insert Values (press Enter to finish) : ";
                do
                {
                        cin >> num;
                        QHead = QueuePush(QHead, num);
                        ch = getchar();
                } while (ch != '\n');
                displayQueue(QHead);
                break;
        case 4:
                cout << "\nQueue Pop -------------------\n";
                QHead = QueuePop(QHead);
                displayQueue(QHead);
                break;
        default:
                break;
        }
    } while (selection != 5);

    cout << "\n\nEND OF PROGRAM\n\n";
}

StackNode* StackPush(StackNode* head, float num)
{
    StackNode* newNode, * nodePtr;

    //Allocate a new stack
    //Store num
    newNode = new StackNode;
    newNode->value = num;
    newNode->next = NULL;
    newNode->prev = NULL;

    if (head == NULL)   //If no node, make newNode the first stack
        head = newNode;
    else   //otherwise make num go to the end
    {
        nodePtr = head;
        // Find the last node and then insert
        while (nodePtr->next != NULL)
            nodePtr = nodePtr->next;
        nodePtr->next = newNode;
        newNode->prev = nodePtr;
    }
    return head;
}
StackNode* StackPop(StackNode* head)
{
    StackNode* nodePtr;

    if (head == NULL)    // If the list is empty, do nothing.
        return head;
    else   // Otherwise pop the last input value
    {
        nodePtr = head;
        while (nodePtr->next != NULL)
        {
```

```
                        nodePtr = nodePtr->next;
                }
                if (nodePtr->next == NULL)
                {
                        nodePtr->prev->next = nodePtr->next;
                        if (nodePtr->next != NULL)
                                nodePtr->next->prev = nodePtr->prev;
                        delete nodePtr;
                }
        }
        return head;
}
void displayStack(StackNode* head)
{
        StackNode* nodePtr;
        nodePtr = head;
        //Find the last node
        while (nodePtr->next != NULL)
        {
                nodePtr = nodePtr->next;
        }
        //Print from the last input
        do
        {
                cout << nodePtr->value << endl;
                nodePtr = nodePtr->prev;
        } while (nodePtr != NULL);
}

QueueNode* QueuePush(QueueNode* head, float num)
{
        QueueNode* newNode, * nodePtr;

        //Allocate a new stack
        //Store num
        newNode = new QueueNode;
        newNode->value = num;
        newNode->next = NULL;
        newNode->prev = NULL;

        if (head == NULL)   //If no node, make newNode the first stack
                head = newNode;
        else   //otherwise make num go to the end
        {
                nodePtr = head;
                // Find the last node and then insert
                while (nodePtr->next != NULL)
                        nodePtr = nodePtr->next;
                nodePtr->next = newNode;
                newNode->prev = nodePtr;
        }
        return head;
}
QueueNode* QueuePop(QueueNode* head)
{
        QueueNode* nodePtr;
        if (head == NULL)    // If the list is empty, do nothing.
                return head;
        else   // Otherwise pop the first input value
        {
                nodePtr = head;
                nodePtr = head->next;
```

```
                nodePtr->prev = NULL;
                delete head;
                head = nodePtr;
                return head;
        }
}
void displayQueue(QueueNode* head)
{
        QueueNode* nodePtr;

        nodePtr = head;
        //Find the last node
        while (nodePtr->next != NULL)
        {
                nodePtr = nodePtr->next;
        }
        //Print from the last input
        do
        {
                cout << nodePtr->value << endl;
                nodePtr = nodePtr->prev;
        } while (nodePtr != NULL);
}
```

```
1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
1

Stack Push ------------------
Insert Values (press Enter to finish) : 1 3 5 7 9 10
10
9
7
5
3
1

1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
2

Stack Pop ------------------
9
7
5
3
1

1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
3

Queue Push ------------------
Insert Values (press Enter to finish) : 2 1 3 5 7 9
9
7
5
3
1
2
```


1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
4
Queue Pop --------------------
9
7
5
3
1
1. Stack Push
2. Stack Pop
3. Queue Push
4. Queue Pop
5. Exit
5
END OF PROGRAM
C:₩Users₩yooni₩source₩repos₩Assignment2_Q1₩De
To automatically close the console when debu
Press any key to close this window . . .

# Problem Solving and Programming 2

## Binary Tree

```
#include <iostream>
#include <stdio.h>
using namespace std;

//Structure declaration
struct treeNode
{
	int key_value;
	struct treeNode* left;
	struct treeNode* right;
};

//Pointer initialisation
struct treeNode* root = NULL;

//Functions
treeNode* create(treeNode* root);
treeNode* insert(int key, treeNode* root);
bool search(int key, treeNode* root);
void ShowTree(treeNode* temp);
void info(int num, treeNode* root);
treeNode* deleteLeaf(int num, treeNode* root);
treeNode* deleteAll(treeNode* root);

int main()
{
	float num;
	int selection(0);
	root = NULL;

	cout << "Tree Creation : -------------------------\n";
	root = create(root);
	cout << "Created Node : \n";
	ShowTree(root);

	do
	{
		cout << "\n\nSelect One : -------------------------\n";
		cout << "1. Insert\n";
		cout << "2. Search\n";
		cout << "3. Leaf Info\n";
		cout << "4. Delete\n";
		cout << "5. Delete All\n";
		cout << "6. ShowTree Display\n";
		cout << "7. Exit\n";
		cin >> selection;

		switch (selection)
		{
		case 1:
			cout << "\nInsertion : ----------------------\n";
			cout << "Input Number to Insert : ";
			cin >> num;
			root = insert(num, root);
			cout << "END";
			break;
		case 2:
			cout << "\nSearch : -------------------------\n";
			cout << "Input Number to Search : ";
			cin >> num;
```

```
                        if (!search(num, root))
                                cout << "Does not exist\n";
                        else
                                cout << "Does exist\n";
                        cout << "END";
                        break;
                case 3:
                        cout << "\nLeaf Info : ----------------------\n";
                        cout << "Input Number for Info : ";
                        cin >> num;
                        if (search(num, root))
                                info(num, root);
                        else
                                cout << "INVALID ERROR\n";
                        cout << "END";
                        break;
                case 4:
                        cout << "\nDelete Leaf : ----------------------\n";
                        cout << "Input Number to Delete : ";
                        cin >> num;
                        if (search(num, root))
                                root = deleteLeaf(num, root);
                        else
                                cout << "INVALID ERROR\n";
                        cout << "END";
                        break;
                case 5:
                        cout << "\nDelete All : ----------------------\n";
                        root = deleteAll(root);
                        cout << "END";
                        break;
                case 6:
                        cout << "\nTree Display : ----------------------\n";
                        ShowTree(root);
                        cout << "END";
                        break;
                default:
                        break;
                }
        } while (selection != 7);

        cout << "\nEnd of the Binary Tree \n";

        return 0;
}

treeNode* create(treeNode* root)
{
        float num;
        char ch;
        cout << "Insert Values (press Enter to finish) : ";
        do
        {
                cin >> num;
                //Input values create a binary tree in order
                root = insert(num, root);
                ch = getchar();
        } while (ch != '\n');
        return root;
}
treeNode* insert(int key, treeNode* root)
{
```

```
	treeNode* leaf, * ptr, * prevptr = NULL;

	//Allocate a new leaf
	//Store num
	leaf = new treeNode;
	leaf->key_value = key;
	leaf->left = NULL;
	leaf->right = NULL;

	if (root == NULL) //If no leaf, make new leaf the root
		root = leaf;
	else
	{
		ptr = root;
		while (ptr != NULL)
		{   //Find the key's place where there isn't a leaf
			//by moving left and right from the root
			if (key < ptr->key_value)
			{
				prevptr = ptr; ptr = ptr->left;
			}
			else
			{
				prevptr = ptr; ptr = ptr->right;
			}
		}
		//Decide to be left or right leaf
		if (key < prevptr->key_value) prevptr->left = leaf;
		else prevptr->right = leaf;
	}
	return root;
}
bool search(int key, treeNode* root)
{
	treeNode* ptr;

	//Initialise boolean value
	bool found = false;

	if (root != NULL)
	{
		ptr = root;
		//Search for the key while
		//ptr has a value but key isn't found
		while (ptr != NULL && !found)
		{
			if (key == ptr->key_value)
			{	//if found, stop
				found = true;
				break;
			}
			else
			{	//Compare the key with the leaf
				//decide whether to go left or right
				if (key < ptr->key_value)
					ptr = ptr->left;
				else
					ptr = ptr->right;
			}
		}
	}
	return found;
```

```
}
void ShowTree(treeNode* temp)
{
	if (temp != NULL) //If there's a value
	{	//Display in ascending order
		ShowTree(temp->left);
		cout << temp->key_value << " ";
		ShowTree(temp->right);
	}
}
void info(int num, treeNode* root)
{
	treeNode* ptr = NULL;
	treeNode* parent = NULL;
	//Initialise boolean for searching
	bool found = false;

	if (search(num, root)) //If num is in the tree
	{
		ptr = root;
		while (ptr != NULL && !found)
			{
				if (num == ptr->key_value)   //If num found from the tree
				{
					//To exit the code later
					found = true;
					//Parent info
					cout << "Parent : ";
					cout << parent->key_value << endl;
					//Left info
					cout << "Left : ";
					if (ptr->left != NULL)
						cout << ptr->left->key_value << endl;
					else
						cout << "None" << endl;
					//Right info
					cout << "Right : ";
					if (ptr->right != NULL)
						cout << ptr->right->key_value << endl;
					else
						cout << "Right : None" << endl;
				}
				else   //Finding the number from the tree
				{
					if (num < ptr->key_value)
					{
						parent = ptr;
						ptr = ptr->left;
					}
					else
					{
						parent = ptr;
						ptr = ptr->right;
					}
				}
			}
	}
	else
	cout << "INVALID ERROR";
}
treeNode* deleteLeaf(int num, treeNode* root)
{
```

```
treeNode* prev, * leaf, * next, * ptrPrev, * ptr;
prev = NULL;
leaf = root;

//Find a leaf which has num i.e. searching
while (leaf != NULL && leaf->key_value != num)
{
	prev = leaf;
	if (num < prev->key_value)
		leaf = prev->left;
	else
		leaf = prev->right;
}
//If num's not in tree
if (!leaf)
{
	cout << "\nINVALID ERROR - Input value not in BT\n";
}

//NUM FOUND SCENARIO FROM HERE
//If the key's leaf has 0 node
if (leaf->left == NULL && leaf->right == NULL)
{
	if (prev->left == leaf)
		prev->left = NULL;
	else
		prev->right = NULL;
}
//If the key's leaf has 1 node
else if (leaf->left == NULL || leaf->right == NULL)
{
	if (leaf->left != NULL)
		next = leaf->left;
	else
		next = leaf->right;
	if (prev)
	{
		if (prev->left == leaf)
			prev->left = next;
		else
			prev->right = next;
	}
}
//If the key's leaf has 2 node
else
{
	//Find the smallest value from right sub-tree
	ptrPrev = leaf;
	ptr = leaf->right;
	while (ptr->left != NULL)
	{
		ptrPrev = ptr;
		ptr = ptr->left;
	}
	//If the smallest values has a right node
	//link this node with the previous node
	if (ptr->right != NULL)
		ptrPrev->left = ptr->right;
	//
	leaf->key_value = ptr->key_value;
}
```

```
        return (root);
}
treeNode* deleteAll(treeNode* root)
{
        treeNode* Ptr, * nextR, *nextL;
        Ptr = root;
        //Cascading NULL values to delete all
        while (Ptr !=NULL)
        {
                nextR = Ptr->right;
                nextL = Ptr->left;
                delete Ptr;
                if (!nextL)   //If root's left leaf is not NULL, delete all left first
                        Ptr = nextL;
                else   //Then delete all right leaves
                        Ptr = nextR;
        }
        root = Ptr;
        return root;
}
```

```
Tree Creation : -------------------------
Insert Values (press Enter to finish) : 35 3 12 7 18 26 22 30 23 68 99
Created Node :
3 7 12 18 22 23 26 30 35 68 99

Select One : -------------------------
1. Insert
2. Search
3. Leaf Info
4. Delete
5. Delete All
6. Inorder Display
7. Exit
1

Insertion : ----------------------
Input Number to Insert : 60
END

Select One : -------------------------
1. Insert
2. Search
3. Leaf Info
4. Delete
5. Delete All
6. Inorder Display
7. Exit
2

Search : -------------------------
Input Number to Search : 99
Does exist
END

Select One : -------------------------
1. Insert
2. Search
3. Leaf Info
4. Delete
5. Delete All
6. Inorder Display
7. Exit
3
```

```
Select One : ------------------------
1. Insert
2. Search
3. Leaf Info
4. Delete
5. Delete All
6. Inorder Display
7. Exit
3

Leaf Info : ----------------------
Input Number for Info : 26
Parent : 18
Left : 22
Right : 30
END

Select One : ------------------------
1. Insert
2. Search
3. Leaf Info
4. Delete
5. Delete All
6. Inorder Display
7. Exit
4

Delete Leaf : ----------------------
Input Number to Delete : 18
END

Select One : ------------------------
1. Insert
2. Search
3. Leaf Info
4. Delete
5. Delete All
6. Inorder Display
7. Exit
6

Tree Display : ----------------------
3 7 12 22 23 26 30 35 60 68 99 END
```

# Problem Solving and Programming 2