

ACTIVITY NO. 3**LINKED LISTS**

LINKED LISTS	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed:
Section: CPE21S4	Date Submitted:
Name: Cruz, Axl Waltz E.	Instructor: Engr. Jimlord Quejado
1. Objective(s)	
<ul style="list-style-type: none">• Implement the list ADT using singly and doubly linked lists• Define operations based on list ADT from the module discussion	
2. Intended Learning Outcomes (ILOs)	
After this activity, the student should be able to: <ul style="list-style-type: none">a. Construct C++ code for a singly and doubly linked list in C++b. Solve given problems utilizing linked lists in C++	
3. Discussion	

PART A: What is a linked list?

Linked Lists are a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

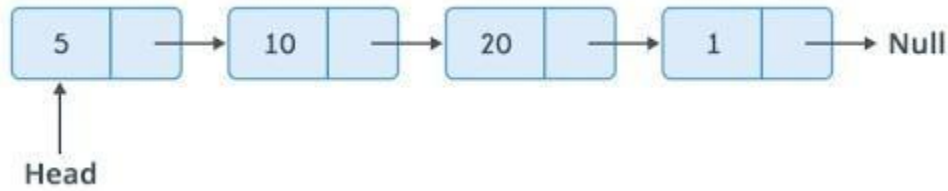


Image Source: Hackearth.com

Why Linked List?

- Arrays are useful but have the following limitations:
- Fixed size.
- Allocated memory remains to be the upper limit.
- Expensive operations (such as insertion) which requires movement of all existing elements and creation of room for new elements.

Advantages over Arrays

- Dynamic size
- Ease of insertion/deletion

Drawbacks

- Random access is not allowed. We have to access elements sequentially starting from the first node
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. No locality reference.

Representation:

- A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.
- Each node in a list consists of at least two parts:
 - Data
 - Pointer (Or Reference) to the next node

PART B: Doubly Linked Lists

Doubly Linked Lists are traversed in either direction. It is a linked list in which every node has a next pointer and a backpointer.

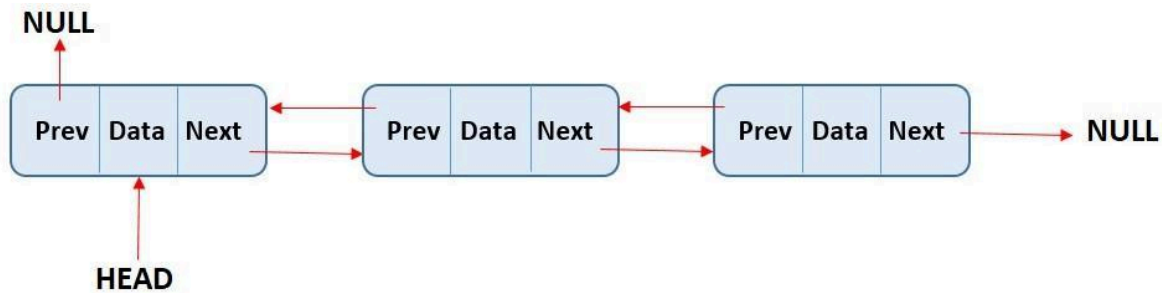


Image Source: AlphaCodingSkills

Every node contains address of next node (except the last node). Every node contains address of previous node (except the first node).

PART C: Common Operations on

Linked Lists Typical operations:

- Initialize the list
- Destroy the list
- Determine if list empty
- Search list for a given item
- Insert an item
- Delete an item, and so on

4. Materials and Equipment

Personal Computer with C++ IDE

Recommended IDE:

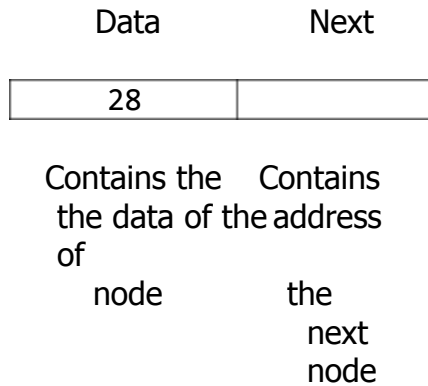
- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A: Construct C++ code for a singly and doubly linked list in C++

A.1. Singly Linked List

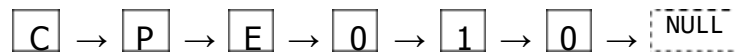
To start, we will do a simple implementation of a linked list. We must keep in mind the visual representation of the individual nodes that will make up our linked list:



Every node in a singly linked list will have 2 compartments, the data and the pointer to the next. The data contains the element of a single type that we want it to contain. The link will point to the next node in our list. In C++, the node is defined as:

```
class Node{
public:
    char data;
    Node *next;
};
```

We will implement a list to represent the string "CPE010" that looks like the figure below:



Implementation will follow the given steps:

1. Create the node pointers and initialize as NULL
2. Create new instances of the node class and allocate them in the heap
3. Define every node in the list
4. Point the last node

to null Simple

implementation:

```
#include<iostream>
#include<utility>

class Node{
public:
    char data;
    Node *next;
```

```
};
```

```
int main(){  
    //step 1  
    Node *head = NULL;
```



```

Node *second = NULL;
Node *third = NULL;
Node *fourth = NULL;
Node *fifth = NULL;
Node *last = NULL;

//step 2
head = new Node;
second = new Node;
third = new Node;
fourth = new Node;
fifth = new Node;
last = new Node;

//step 3
head->data = 'C';
head->next = second;

second->data = 'P';
second->next = third;

third->data = 'E';
third->next = fourth;

fourth->data = 'O';
fourth->next = fifth;

fifth->data = 'I';
fifth->next = last;

//step 4
last->data = '0';
last->next = nullptr;
}

```

Although we have created the linked list, this is an implementation that is useless and meant only to show what we want to happen for the given output. **Run your code and screenshot the output, then briefly discuss how the output came to be and how could it be improved (table 3-1)?**

Imagine if we had to make multiple items in the list in the hundreds or thousands! This would be too tedious, ineffective, and inefficient. So, we must implement certain methods. These methods/operations associated with the linked lists are:

- Traversal
- Insertion at head
- Insertion at any part of the list
- Insertion at the end
- Deletion of a

node Linked List

Traversal

Algorithm: ListTraversal (parameter: pointer to node n)


```
WHILE n IS NOT EQUAL TO null  
    PRINT data OF n  
    GO TO NEXT NODE n := next  
ENDWHILE
```

```
1 // Linked List Implementation
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data;
7     Node* next;
8 }
9
10 Node* head = NULL;
11 Node* n = NULL;
12
13 void insertAtFront(Node*& head, char v) {
14     Node* n = new Node(v);
15     if (!head) {
16         head = n;
17         return;
18     }
19     Node* t = head;
20     while (t->next != NULL) {
21         t = t->next;
22     }
23     t->next = n;
24 }
25
26 void insertAtBack(Node*& head, char v) {
27     Node* n = new Node(v);
28     head = n;
29 }
30
31 void insertAtPos(Node*& head, char v, int pos) {
32     if (!head) {
33         head = n;
34         return;
35     }
36     Node* t = head;
37     while (t->next != NULL) {
38         t = t->next;
39     }
40     t->next = n;
41 }
42
43 void deleteNode(Node*& head, char key) {
44     if (!head) {
45         return;
46     }
47     Node* t = head;
48     while (t->next != NULL) {
49         if (t->data == key) {
50             delete t;
51             t = t->next;
52         }
53         t = t->next;
54     }
55 }
```

```
PRINT next line
END
```

Sometimes we have a linked list, and we need to insert a node somewhere other than at the end of the list. We will look at a couple of different ways to insert a node into an existing list.

To insert a node at the head:

1. Allocate memory for the new node
2. Put our data into the new node
3. Set Next of the new node to point to the previous Head
4. Reset Head to point to the new node

To insert a node at any location between the head and tail:

1. Check if it is the head node (previous node is null)
2. If null, print "Previous node cannot be null."
3. Allocate a new node
4. Store data in the new node
5. Point new node to the node previous node was pointing to
6. Point previous node to the new node

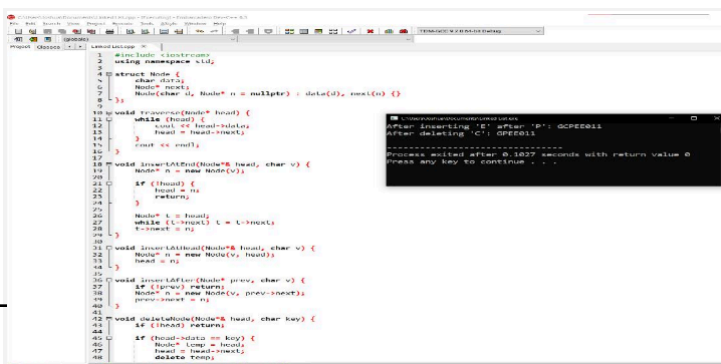
To insert a node at the end:

1. Allocate new node
2. Dereference to the head node
3. Store data in new node
4. Point next of new node to NULL
5. Traverse the list until next of the node is null
6. Point the next of the current node to the new node

To delete a node from linked list:

1. Find previous node of the node to be deleted.
2. Change the next of previous node.
3. Free memory for the node to be deleted.

Create code for each of the pseudocode given for all list operations above. Provide screenshots in table 3-2.



The screenshot shows a C++ code editor with the following code:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8 Node* head = NULL;
9 Node* tail = NULL;
10 Node* createNode(char v) {
11     Node* n = new Node(v);
12     return n;
13 }
14 void insertAtHead(Node* head, char v) {
15     Node* n = new Node(v);
16     if (head == NULL) {
17         head = n;
18         return;
19     }
20     Node* t = head;
21     while (t->next != NULL) {
22         t = t->next;
23     }
24     t->next = n;
25 }
26 void insertAtTail(Node* head, char v) {
27     Node* n = new Node(v);
28     head = n;
29 }
30 void insertAtPos(Node* prev, char v) {
31     if (prev == NULL) {
32         return;
33     }
34     Node* n = new Node(v);
35     prev->next = n;
36 }
37 void deleteNode(Node* head, char key) {
38     if (head == NULL) {
39         return;
40     }
41     if (head->data == key) {
42         Node* temp = head;
43         head = head->next;
44         delete temp;
45     }
46 }
```

The terminal window shows the output of the program:

```
After inserting 'E' after 'P': GEEDBLL
After deleting 'C': GDEEDLL
Press any key to continue . . .
```

In your driver function, show the use of each list operation and show the output in table 3-3 found in section 6 with a descriptive caption for each. The tasks you have to perform are as follows:

- Traverse the list by passing the head of the created list into the function
- Insert the element 'G' at the start of the list to replace the current node. Output should now show "GCPE101"
- Insert an element "E" with the previous node element being "P". Output should now show "GCPEE101".
- Delete the node containing the element C.

The screenshot shows a C++ IDE with a source code editor on the left and a console window on the right. The source code implements a singly linked list with functions for traversal, insertion at the start, insertion before a specific node, and deletion of a node. The console output shows the results of these operations: initial list "CPE101", insertion of 'G' resulting in "GCPE101", insertion of 'E' before 'P' resulting in "GCPEE101", and deletion of 'C' resulting in "GPEE101".

```

1 // Driver program to test basic operations
2 #include <iostream>
3 using namespace std;
4
5 // Structure of a node
6 struct Node {
7     char data;
8     Node* next;
9 };
10 Node* head = NULL;
11
12 // Function to create a new node
13 Node* createNode(char data) {
14     Node* n = new Node;
15     n->data = data;
16     n->next = NULL;
17     return n;
18 }
19
20 // Function to insert a new node at the start
21 void insertAtStart(Node* head, char v) {
22     Node* n = createNode(v);
23     n->next = head;
24     head = n;
25 }
26
27 // Function to insert a new node before a specific node
28 void insertBefore(Node* head, char v, char key) {
29     Node* n = createNode(v);
30     Node* temp = head;
31     while (temp->data != key) {
32         temp = temp->next;
33     }
34     n->next = temp;
35     temp = head;
36 }
37
38 // Function to delete a node
39 void deleteNode(Node* head, char key) {
40     Node* temp = head;
41     Node* prev = NULL;
42     while (temp->data != key) {
43         prev = temp;
44         temp = temp->next;
45     }
46     if (prev == NULL) {
47         head = temp->next;
48     } else {
49         prev->next = temp->next;
50     }
51     delete temp;
52 }
53
54 // Function to traverse the list
55 void traverseList(Node* head) {
56     while (head != NULL) {
57         cout << head->data << " ";
58         head = head->next;
59     }
60     cout << endl;
61 }
62
63 // Driver code
64 int main() {
65     head = createNode('C');
66     insertAtStart(head, 'G');
67     insertBefore(head, 'E', 'P');
68     deleteNode(head, 'C');
69     traverseList(head);
70     return 0;
71 }

```

```

=====
After inserting 'E' after 'P': GCPEE101
After deleting 'C': GPEE101
=====
Process exited after 0.1027 seconds with return value 0
Press any key to continue . . .

```

- e. Delete the node containing the element P.
- f. Show the elements in the list. Output should be "GEE101".

A.2. Doubly Linked List

The singly linked list allows for direct access from a list node only to the next node in the list. A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list.

The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.

Prev	Data	Next
←	28	→

Contains the address of the previous node

Contains the data of the node

Contains the address of the next node

```

1 // doubly linked list program
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data;
7     Node* prev;
8     Node* next;
9 };
10 Node* head = NULL;
11 Node* tail = NULL;
12
13 void insertAtFront(Node*& head, char v) {
14     Node* n = new Node(v);
15     if (!head) {
16         head = n;
17         tail = n;
18     } else {
19         n->next = head;
20         head->prev = n;
21         head = n;
22     }
23 }
24
25 void insertAtBack(Node*& head, char v) {
26     Node* n = new Node(v);
27     if (!tail) {
28         head = n;
29         tail = n;
30     } else {
31         n->prev = tail;
32         tail->next = n;
33         tail = n;
34     }
35 }
36
37 void deleteNode(Node*& head, char key) {
38     if (!head) return;
39     if (head->data == key) {
40         head = head->next;
41         delete head;
42     } else {
43         Node* temp = head;
44         while (temp->next != NULL) {
45             if (temp->next->data == key) {
46                 temp->next = temp->next->next;
47                 delete temp->next;
48             }
49             temp = temp->next;
50         }
51     }
52 }
53
54 int main() {
55     insertAtFront(head, 'G');
56     insertAtFront(head, 'E');
57     insertAtFront(head, 'E');
58     insertAtBack(head, '1');
59     insertAtBack(head, '0');
60     insertAtBack(head, '1');
61     deleteNode(head, 'E');
62     cout << "List after deletion: ";
63     displayList(head);
64     return 0;
65 }

```

```

Output:
List after deletion: GEE101
Press any key to continue . . .

```

s
o
f
t
h

e
n
e
x
t

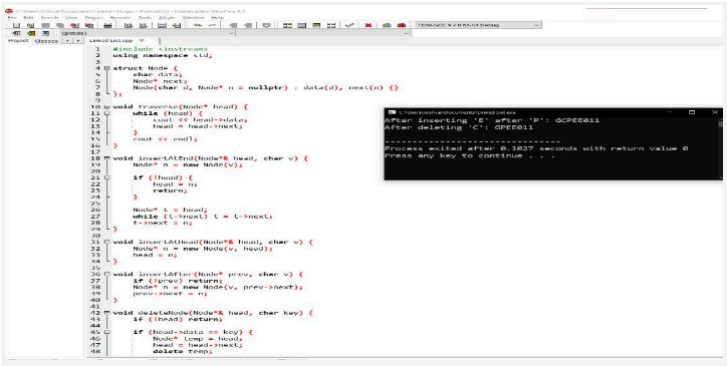
n
o
d
e

This means that in addition to our implementation of the singly linked list, we'll have addition compartment.

```
class Node{
public:
    char data;
    Node *next;
    Node *prev;
};
```

Modify the given operations used in the singly linked lists to work on the new construct of a doubly linked list. Provide a screenshot of your code and analysis in table 3-4.

6. Output



Screenshot	<pre>#include <iostream> using namespace std; // Only define Node once class Node { public: char data; Node *next; }; int main() { // Step 1 Node *head = NULL; Node *second = NULL; Node *third = NULL; Node *fourth = NULL; Node *fifth = NULL; Node *last = NULL; // Step 2 head = new Node; second = new Node; third = new Node; fourth = new Node; fifth = new Node; last = new Node; // Step 3 head->data = 'C'; head->next = second; second->data = 'P'; second->next = third; third->data = 'E'; third->next = fourth; fourth->data = 'O'; fourth->next = fifth; fifth->data = 'I'; fifth->next = last; // Step 4 last->data = '\0'; last->next = nullptr; // Optional: Print to verify (not required if you don't want changes) Node* temp = head; while (temp != nullptr) { cout << temp->data << " "; temp = temp->next; } cout << endl; return 0; }</pre>	<div>STDIN</div> <div>Input for the program (Optional)</div> <hr/> <div>Output:</div> <div>C P E O I \0</div>
Discussion	<p>This list is properly constructed, each node is correctly linked and the data forms the string when read sequentially, to improve the code encapsulation with a link list could do and memory management which is a destructor to allocate memory and never freeing it.</p>	

Table 3-1. Output of Initial/Simple Implementation

Operation	Screenshot
-----------	------------

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

struct Node {
    char data;
    Node *next;
};

Node* head = NULL;
Node* tail = NULL;

void insertAtFront(Node* head, char v) {
    Node* n = new Node(v);
    if (!head) {
        head = n;
        tail = n;
        return;
    }
    n->next = head;
    head = n;
}

void insertAtBack(Node* head, char v) {
    Node* n = new Node(v);
    if (!tail) {
        head = n;
        tail = n;
        return;
    }
    tail->next = n;
    tail = n;
}

void deleteNode(Node* head, char key) {
    if (!head) return;
    if (head->data == key) {
        head = head->next;
        delete head;
    }
    else {
        Node* temp = head;
        while (temp->next != NULL) {
            if (temp->next->data == key) {
                temp->next = temp->next->next;
                delete temp->next;
            }
            temp = temp->next;
        }
    }
}
```

Output:

```
After inserting 'E' after 'P': CPEOI
After deleting 'C': DEOI
Press any key to continue . . .
```

Traversal	<pre>// 1. Traversal void traverse(Node* head) { Node* current = head; while (current != nullptr) { cout << current->data << " "; current = current->next; } cout << endl; }</pre>
Insertion at head	<pre>// 2. Insertion at head void insertAtHead(Node*& head, char value) { Node* newNode = new Node; newNode->data = value; newNode->next = head; head = newNode; }</pre>
Insertion at any part of the list	<pre>// 3. Insertion at any position (0-based) void insertAtPosition(Node*& head, int position, char value) { if (position == 0) { insertAtHead(head, value); return; } }</pre> <p>-</p>

```

1 // 1. Traversal
2 void traverse(Node* head) {
3     Node* current = head;
4     while (current != nullptr) {
5         cout << current->data << " ";
6         current = current->next;
7     }
8     cout << endl;
9 }
10
11 // 2. Insertion at head
12 void insertAtHead(Node*& head, char value) {
13     Node* newNode = new Node;
14     newNode->data = value;
15     newNode->next = head;
16     head = newNode;
17 }
18
19 // 3. Insertion at any position (0-based)
20 void insertAtPosition(Node*& head, int position, char value) {
21     if (position == 0) {
22         insertAtHead(head, value);
23         return;
24     }
25 }
26
27 // 4. Deletion
28 void deleteNode(Node*& head, char key) {
29     if (head == nullptr) return;
30     if (head->data == key) {
31         Node* temp = head;
32         head = head->next;
33         delete temp;
34     }
35     else {
36         Node* prev = head;
37         while (prev->next != nullptr) {
38             if (prev->next->data == key) {
39                 prev->next = prev->next->next;
40                 delete prev->next;
41             }
42             prev = prev->next;
43         }
44     }
45 }
46
47 int main() {
48     Node* head = nullptr;
49     char ch;
50     int pos;
51     while (ch != 'q') {
52         cout << "1. Insert at head\n";
53         cout << "2. Insert at position\n";
54         cout << "3. Delete\n";
55         cout << "4. Traverse\n";
56         cout << "5. Exit\n";
57         cout << "Enter choice: ";
58         int choice;
59         while (choice < 1 || choice > 5) {
60             choice = 0;
61             cout << "Invalid choice. Enter again: ";
62         }
63         switch (choice) {
64             case 1: {
65                 char value;
66                 while (value != '\n') {
67                     value = getche();
68                 }
69                 insertAtHead(head, value);
70                 break;
71             }
72             case 2: {
73                 int position;
74                 while (position < 0 || position > 100) {
75                     position = 0;
76                     cout << "Invalid position. Enter again: ";
77                 }
78                 char value;
79                 while (value != '\n') {
80                     value = getche();
81                 }
82                 insertAtPosition(head, position, value);
83                 break;
84             }
85             case 3: {
86                 char key;
87                 while (key != '\n') {
88                     key = getche();
89                 }
90                 deleteNode(head, key);
91                 break;
92             }
93             case 4: {
94                 traverse(head);
95                 break;
96             }
97             case 5: {
98                 return 0;
99             }
100          }
101      }
102      return 0;
103  }

```

Output in terminal:

```

After inserting 'E' after 'P': GCPEDB1
After deleting 'C': GPEDB1
.....
Process exited after 0.1027 seconds with return value 0
Press any key to continue . . .

```


Insertion at the end

```
// 4. Insertion at end
void insertAtEnd(Node*& head, char value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
}
```

```

1 // 4. Insertion at end
2 void insertAtEnd(Node*& head, char value) {
3     Node* newNode = new Node;
4     newNode->data = value;
5     newNode->next = nullptr;
6
7     if (head == nullptr) {
8         head = newNode;
9         return;
10    }
11
12    Node* temp = head;
13    while (temp->next != nullptr) {
14        temp = temp->next;
15    }
16
17    temp->next = newNode;
18 }
19
20 // 5. Insertion at position
21 void insertAtPos(Node*& head, char v, int pos) {
22     if (pos < 1) return;
23     if (pos == 1) {
24         insertAtFront(head, v);
25         return;
26     }
27     Node* temp = head;
28     while (temp->next != nullptr) {
29         temp = temp->next;
30     }
31     insertAtEnd(head, v);
32 }
33
34 // 6. Deletion of a node
35 void deleteNode(Node*& head, char key) {
36     if (head == nullptr) return;
37     if (head->data == key) {
38         head = head->next;
39         delete head;
40     }
41     Node* temp = head;
42     while (temp->next != nullptr) {
43         if (temp->next->data == key) {
44             temp->next = temp->next->next;
45             delete temp->next;
46         }
47         temp = temp->next;
48     }
49 }
50
51 int main() {
52     Node* head = nullptr;
53     insertAtFront(head, 'G');
54     insertAtFront(head, 'O');
55     insertAtFront(head, 'P');
56     insertAtEnd(head, 'E');
57     insertAtPos(head, 'D', 4);
58     insertAtPos(head, 'S', 6);
59     display(head);
60     deleteNode(head, 'D');
61     display(head);
62     deleteNode(head, 'S');
63     display(head);
64     return 0;
65 }

```

Output:

```

After inserting 'E' after 'P': G O P E D S
After deleting 'D': G O P E S
Press any key to continue . . .

```

Deletion of a node

```
// 5. Deletion of a node by value
void deleteNode(Node*& head, char value) {
    if (head == nullptr) return;

    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

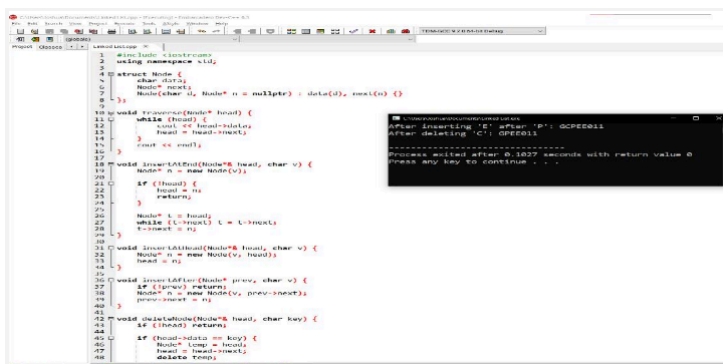
    Node* prev = head;
    Node* curr = head->next;

    while (curr != nullptr && curr->data != value) {
        prev = curr;
        curr = curr->next;
    }

    if (curr == nullptr) {
        cout << "Value not found." << endl;
        return;
    }

    prev->next = curr->next;
    delete curr;
}
```

Table 3-2. Code for the List Operations



Creates a new node with its next pointer referencing the current head, then updates the head to this new node.

Analysis: It locates the node “P,” creates a new node, a

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8
9 Node* head = nullptr;
10 Node* n = nullptr;
11
12 void traverse(Node* head) {
13     while (head) {
14         cout << head->data;
15         head = head->next;
16     }
17     cout << endl;
18 }
19
20 void insertAtEnd(Node* head, char v) {
21     Node* n = new Node(v);
22     if (!head) {
23         head = n;
24         return;
25     }
26     Node* t = head;
27     while (t->next) {
28         t = t->next;
29     }
30     t->next = n;
31 }
32
33 void insertAfter(Node* head, char v) {
34     Node* n = new Node(v, head);
35     head = n;
36 }
37
38 void insertAfter(Node* prev, char v) {
39     if (!prev) return;
40     Node* n = new Node(v, prev->next);
41     prev->next = n;
42 }
43
44 int main() {
45     Node* head = nullptr;
46     insertAtEnd(head, 'C');
47     insertAtEnd(head, 'P');
48     insertAtEnd(head, 'Q');
49     traverse(head);
50 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8
9 Node* head = nullptr;
10 Node* n = nullptr;
11
12 void traverse(Node* head) {
13     while (head) {
14         cout << head->data;
15         head = head->next;
16     }
17     cout << endl;
18 }
19
20 void insertAtEnd(Node* head, char v) {
21     Node* n = new Node(v);
22     if (!head) {
23         head = n;
24         return;
25     }
26     Node* t = head;
27     while (t->next) {
28         t = t->next;
29     }
30     t->next = n;
31 }
32
33 void insertAfter(Node* head, char v) {
34     Node* n = new Node(v, head);
35     head = n;
36 }
37
38 void insertAfter(Node* prev, char v) {
39     if (!prev) return;
40     Node* n = new Node(v, prev->next);
41     prev->next = n;
42 }
43
44 void deleteNode(Node* head, char key) {
45     if (!head) return;
46     if (head->data == key) {
47         Node* temp = head;
48         head = head->next;
49         delete temp;
50     }
51 }
52
53 int main() {
54     Node* head = nullptr;
55     insertAtEnd(head, 'C');
56     insertAtEnd(head, 'P');
57     insertAtEnd(head, 'Q');
58     traverse(head);
59 }

```

Table 3-3. Code and Analysis for Singly Linked L

The function moves through the list by advancing

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8
9 Node* head = nullptr;
10 Node* n = nullptr;
11
12 void traverse(Node* head) {
13     while (head) {
14         cout << head->data;
15         head = head->next;
16     }
17     cout << endl;
18 }
19
20 void insertAtEnd(Node* head, char v) {
21     Node* n = new Node(v);
22     if (!head) {
23         head = n;
24         return;
25     }
26     Node* t = head;
27     while (t->next) {
28         t = t->next;
29     }
30     t->next = n;
31 }
32
33 void insertAfter(Node* head, char v) {
34     Node* n = new Node(v, head);
35     head = n;
36 }
37
38 void insertAfter(Node* prev, char v) {
39     if (!prev) return;
40     Node* n = new Node(v, prev->next);
41     prev->next = n;
42 }
43
44 int main() {
45     Node* head = nullptr;
46     insertAtEnd(head, 'C');
47     insertAtEnd(head, 'P');
48     insertAtEnd(head, 'Q');
49     traverse(head);
50 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8
9 Node* head = nullptr;
10 Node* n = nullptr;
11
12 void traverse(Node* head) {
13     while (head) {
14         cout << head->data;
15         head = head->next;
16     }
17     cout << endl;
18 }
19
20 void insertAtEnd(Node* head, char v) {
21     Node* n = new Node(v);
22     if (!head) {
23         head = n;
24         return;
25     }
26     Node* t = head;
27     while (t->next) {
28         t = t->next;
29     }
30     t->next = n;
31 }
32
33 void insertAfter(Node* head, char v) {
34     Node* n = new Node(v, head);
35     head = n;
36 }
37
38 void insertAfter(Node* prev, char v) {
39     if (!prev) return;
40     Node* n = new Node(v, prev->next);
41     prev->next = n;
42 }
43
44 void deleteNode(Node* head, char key) {
45     if (!head) return;
46     if (head->data == key) {
47         Node* temp = head;
48         head = head->next;
49         delete temp;
50     }
51 }
52
53 int main() {
54     Node* head = nullptr;
55     insertAtEnd(head, 'C');
56     insertAtEnd(head, 'P');
57     insertAtEnd(head, 'Q');
58     traverse(head);
59 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8
9 Node* head = nullptr;
10 Node* n = nullptr;
11
12 void traverse(Node* head) {
13     while (head) {
14         cout << head->data;
15         head = head->next;
16     }
17     cout << endl;
18 }
19
20 void insertAtEnd(Node* head, char v) {
21     Node* n = new Node(v);
22     if (!head) {
23         head = n;
24         return;
25     }
26     Node* t = head;
27     while (t->next) {
28         t = t->next;
29     }
30     t->next = n;
31 }
32
33 void insertAfter(Node* head, char v) {
34     Node* n = new Node(v, head);
35     head = n;
36 }
37
38 void insertAfter(Node* prev, char v) {
39     if (!prev) return;
40     Node* n = new Node(v, prev->next);
41     prev->next = n;
42 }
43
44 void deleteNode(Node* head, char key) {
45     if (!head) return;
46     if (head->data == key) {
47         Node* temp = head;
48         head = head->next;
49         delete temp;
50     }
51 }
52
53 int main() {
54     Node* head = nullptr;
55     insertAtEnd(head, 'C');
56     insertAtEnd(head, 'P');
57     insertAtEnd(head, 'Q');
58     traverse(head);
59 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* next;
7 };
8
9 void traverse(Node* head) {
10     while (head) {
11         cout << head->data;
12         head = head->next;
13     }
14     cout << endl;
15 }
16
17 int main() {
18     Node* head = nullptr;
19     head = new Node();
20     head->data = 'G';
21     head->next = new Node();
22     head->next->data = 'E';
23     head->next->next = new Node();
24     head->next->next->data = 'E';
25     head->next->next->next = new Node();
26     head->next->next->next->data = 'I';
27     head->next->next->next->next = new Node();
28     head->next->next->next->next->data = 'I';
29     head->next->next->next->next->next = new Node();
30     head->next->next->next->next->next->data = 'I';
31     head->next->next->next->next->next->next = nullptr;
32     cout << "Final list: ";
33     traverse(head);
34     return 0;
35 }

```

```

Final list: GEEIIOI
Process exited after 0.00857 seconds with return value 0
Press any key to continue . . .

```

Table 3-4. Modified Operations for Doubly Linked

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* prev;
7     Node* next;
8 };
9
10 void traverse(Node* head) {
11     while (head) {
12         cout << head->data;
13         head = head->next;
14     }
15     cout << endl;
16 }
17
18 void insertAtEnd(Node* head, char v) {
19     Node* n = new Node;
20     n->data = v;
21     n->next = nullptr;
22     n->prev = nullptr;
23     if (!head) {
24         head = n;
25         return;
26     }
27     Node* t = head;
28     while (t->next) {
29         t = t->next;
30     }
31     t->next = n;
32     n->prev = t;
33 }
34
35 void insertAtBeginning(Node* head, char v) {
36     Node* n = new Node;
37     n->prev = nullptr;
38     n->next = head;
39     if (head) head->prev = n;
40     head = n;
41     n->data = v;
42 }
43
44 void insertAfter(Node* head, char prevData, char v) {

```

```

Initial list: GPEEIOI
GPEEIOI
GPEEIOI
GPEEIOI
Final list: GEEIOI
Process exited after 0.00896 seconds with return value 0
Press any key to continue . . .

```

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     char data;
6     Node* prev;
7     Node* next;
8 };
9
10 void traverse(Node* head) {
11     while (head) {
12         cout << head->data;
13         head = head->next;
14     }
15     cout << endl;
16 }
17
18 void insertAtEnd(Node* head, char v) {
19     Node* n = new Node(v);
20     if (!head) {
21         head = n;
22         return;
23     }
24     Node* t = head;
25     while (t->next) {
26         t = t->next;
27     }
28     t->next = n;
29     n->prev = t;
30 }
31
32 void insertAtBeginning(Node* head, char v) {
33     Node* n = new Node(v);
34     n->prev = nullptr;
35     n->next = head;
36     if (head) head->prev = n;
37     head = n;
38 }
39
40 void insertAfter(Node* head, char prevData, char v) {
41     if (!head) return;
42     Node* n = new Node(v, prevData, head->next);
43     prevData->next = n;
44 }
45
46 void deleteNode(Node* head, char key) {
47     if (!head) return;
48     if (head->data == key) {
49         Node* temp = head;
50         head = head->next;
51         delete temp;
52     }
53     while (head->next) {
54         if (head->next->data == key) {
55             Node* temp = head->next;
56             head->next = head->next->next;
57             delete temp;
58         }
59         head = head->next;
60     }

```

```

After inserting 'E' after 'P': GPEEIOI
After deleting 'C': GPEEIOI
Process exited after 0.1027 seconds with return value 0
Press any key to continue . . .

```

7. Supplementary Activity

ILO B: Solve given problems utilizing linked lists in C++

Problem Title: Implementing a Song Playlist using Linked List

Source: Packt Publishing

Problem Description:

In this activity, we'll look at some applications for which a singly linked list is not enough or not convenient. We will build a tweaked version that fits the application. We often encounter cases where we have to customize default implementations, such as when looping songs in a music player or in games where multiple players take a turn one by one in a circle.

These applications have one common property – we traverse the elements of the sequence in a circular fashion. Thus, the node after the last node will be the first node while traversing the list. This is called a circular linked list.

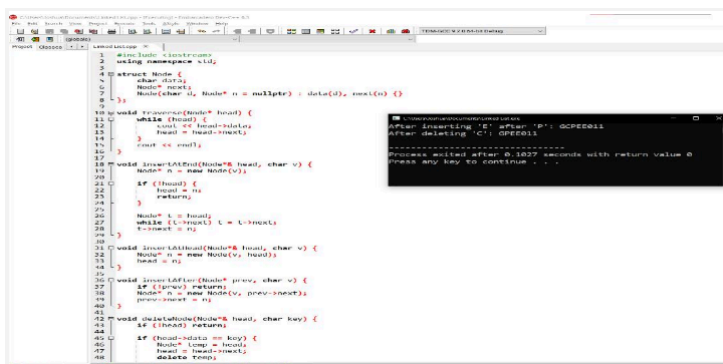
We'll take the use case of a music player. It should have following functions supported:

- Create a playlist using multiple songs.
- Add songs to the playlist.
- Remove a song from the playlist.
- Play songs in a loop (for this activity, we will print all the songs once).

Here are the steps to solve the problem:

- Design the basic structure that supports circular data representation.
- After that, implement the insert and delete functions in the structure.
- Implement a function for traversing the playlist.

The driver function should allow for common operations on a playlist such as: next, previous, play all songs, insert and remove.



```
1 // C++ program to implement a circular linked list
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data;
7     Node* next;
8 };
9
10 Node* createNode(char data) {
11     Node* n = new Node;
12     n->data = data;
13     n->next = NULL;
14     return n;
15 }
16
17 void insertAtEnd(Node* head, char v) {
18     Node* n = new Node(v);
19     if (!head) {
20         head = n;
21         return;
22     }
23     Node* t = head;
24     while (t->next != t) {
25         t = t->next;
26     }
27     t->next = n;
28 }
29
30 void insertAtHead(Node* head, char v) {
31     Node* n = new Node(v);
32     head = n;
33 }
34
35 void insertAtPos(Node* head, char v, int pos) {
36     if (!head) return;
37     Node* n = new Node(v);
38     Node* t = head;
39     for (int i = 1; i < pos; i++) {
40         t = t->next;
41     }
42     t->next = n;
43 }
44
45 void deleteNode(Node* head, char key) {
46     if (!head) return;
47     Node* t = head;
48     while (t->next != t) {
49         if (t->data == key) {
50             t->next = t->next->next;
51             delete t;
52             return;
53         }
54         t = t->next;
55     }
56 }
```

```

#include <string>
using namespace std;

struct Song {
    string title;
    Song* next;
    Song* prev;
};

void addSong(Song*& head, const string& title) {
    Song* newSong = new Song{title, nullptr, nullptr};
    if (!head) {
        newSong->next = newSong;
        newSong->prev = newSong
    }
}

#include <string>
using namespace std;

struct Song {
    string title;
    Song* next;
    Song* prev;
};

void addSong(Song*& head, const string& title) {
    Song* newSong = new Song{title, nullptr, nullptr};
    if (!head) {
        newSong->next = newSong;
        newSong->prev = newSong;
        head = newSong;
        return;
    }
    Song* tail = head->prev;
    tail->next = newSong;
    newSong->prev = tail;
    newSong->next = head;
    head->prev = newSong;
}

```

The screenshot shows a C++ IDE with a source file and a console window. The source file contains a linked list implementation with functions for inserting, deleting, and displaying nodes. The console window shows the output of the program, including the insertion of nodes 'A', 'B', and 'C', and the deletion of node 'B'.

```

1 // Linked List Implementation
2 #include <iostream>
3 using namespace std;
4 struct Node {
5     char data;
6     Node* next;
7     Node(char d, Node* n = nullptr) : data(d), next(n) {}
8 }
9
10 void insertAtHead(Node*& head, char v) {
11     Node* n = new Node(v);
12     if (!head) {
13         head = n;
14         return;
15     }
16     Node* t = head;
17     while (t->next != nullptr) t = t->next;
18     t->next = n;
19 }
20 void insertAtTail(Node*& head, char v) {
21     Node* n = new Node(v, head);
22     head = n;
23 }
24 void insertAtIndex(Node*& head, char v, int i) {
25     if (!head) {
26         head = n;
27         return;
28     }
29     Node* t = head;
30     while (i-- > 0) t = t->next;
31     t->next = n;
32 }
33 void deleteNode(Node*& head, char key) {
34     if (!head) return;
35     if (head->data == key) {
36         head = head->next;
37         delete head;
38     }
39 }
40 void display(Node* head) {
41     if (!head) return;
42     while (head) {
43         cout << head->data << " ";
44         head = head->next;
45     }
46     cout << endl;
47 }
48 int main() {
49     Node* head = nullptr;
50     insertAtHead(head, 'A');
51     insertAtTail(head, 'B');
52     insertAtIndex(head, 'C', 1);
53     display(head);
54     deleteNode(head, 'B');
55     display(head);
56     return 0;
57 }

```

Output:

```

After inserting 'A' after 'B': ACDEB
After deleting 'C': ACDEB
Press any key to continue . . .

```

```

void removeSong(Song*& head, const string& title) {
    if (!head) return;
    Song* curr = head;
    do {
        if (curr->title == title) {
            if (curr->next == curr) {
                delete curr;
                head = nullptr;
                return;
            }
            curr->prev->next = curr->next;
            curr->next->prev = curr->prev;
            if (curr == head) head = curr->next;
            delete curr;
            return;
        }
        curr = curr->next;
    } while (curr != head);
}

void displayPlaylist(Song* head) {
    if (!head) {
        cout << "Playlist is empty.\n";
        return;
    }
    Song* curr = head;
    do {
        cout << "Playing: " << curr->title << endl;
        curr = curr->next;
    } while (curr != head);
}

int main() {
    Song* playlist = nullptr;

    addSong(playlist, "Song A");
    addSong(playlist, "Song B");
    addSong(playlist, "Song C");
    addSong(playlist, "Song D");

    cout << "Initial Playlist:\n";
    displayPlaylist(playlist);

    cout << "\nRemoving Song B...\n";
    removeSong(playlist, "Song B");

    cout << "\nUpdated Playlist:\n";
}

```

The screenshot shows a C++ IDE with the following code:

```

1 // Song.h
2 #ifndef SONG_H
3 #define SONG_H
4
5 #include <iostream>
6 #include <string>
7
8 struct Node {
9     char* title;
10    Node* next;
11    Node* prev;
12};
13
14 Node* createNode(char* title) {
15     Node* n = new Node;
16     n->title = title;
17     n->next = nullptr;
18     n->prev = nullptr;
19     return n;
20}
21
22 void insertAtFront(Node*& head, char* title) {
23     Node* n = createNode(title);
24     if (!head) {
25         head = n;
26         return;
27     }
28     n->next = head;
29     head->prev = n;
30}
31
32 void insertAtBack(Node*& head, char* title) {
33     Node* n = createNode(title);
34     if (!head) {
35         head = n;
36         return;
37     }
38     Node* curr = head;
39     while (curr->next != nullptr) {
40         curr = curr->next;
41     }
42     curr->next = n;
43     n->prev = curr;
44}
45
46 void insertAtIndex(Node*& head, int index, char* title) {
47     if (index < 0) return;
48     if (index == 0) {
49         insertAtFront(head, title);
50         return;
51     }
52     Node* curr = head;
53     for (int i = 0; i < index; i++) {
54         curr = curr->next;
55     }
56     insertAtBack(curr->next, title);
57}
58
59 void deleteNode(Node*& head, char* title) {
60     if (!head) return;
61     Node* curr = head;
62     while (curr->next != nullptr) {
63         if (curr->title == title) {
64             if (curr->prev == nullptr) {
65                 head = curr->next;
66             } else {
67                 curr->prev->next = curr->next;
68             }
69             if (curr->next != nullptr) {
70                 curr->next->prev = curr->prev;
71             }
72             delete curr;
73             return;
74         }
75         curr = curr->next;
76     }
77     if (curr->title == title) {
78         delete curr;
79         head = nullptr;
80     }
81}
82
83 void displayList(Node* head) {
84     if (!head) {
85         cout << "Empty list\n";
86         return;
87     }
88     Node* curr = head;
89     while (curr->next != nullptr) {
90         cout << curr->title << " ";
91         curr = curr->next;
92     }
93     cout << endl;
94}
95
96 int main() {
97     Node* head = nullptr;
98     insertAtFront(head, "Song A");
99     insertAtBack(head, "Song D");
100    insertAtIndex(head, 1, "Song B");
101    insertAtIndex(head, 2, "Song C");
102    displayList(head);
103    deleteNode(head, "Song B");
104    displayList(head);
105    return 0;
106}

```

The output window shows the following text:

```

After inserting 'D' after 'A': A D C B
After deleting 'C': A D B
Press any key to continue . . .

```

```
39     delete curr;
40     return;
41 }
42 curr = curr->next;
43 } while (curr != head);
44 }
45
46 void C:\Users\TIPQC\Documents\U
47
48 Initial Playlist:
49 Playing: Song A
50 Playing: Song B
51 Playing: Song C
52 Playing: Song D
53
54 Removing Song B...
55
56 Updated Playlist:
57 Playing: Song A
58 Playing: Song C
59 Playing: Song D
60
61 -----
62 Process exited after 0.01568 seconds with return value 0
63 Press any key to continue . . . |
64
65
66
67
68
69
70
71
72
73
74
75
76
77
```

Resources Compile Log Debug Find Results Close

Message

C:\Users\TIPQC\Documents\InitialList1.cpp In function 'void addSong(Song* s, const string&):'

8. Conclusion

I learned about the link list on how nodes are made up in data structure where it stores data and a pointer to the next node allowing dynamic memory usage and easy insertion or deletion. Unlike arrays they didn't store elements in contiguous memory like link list do.

Provide the following:

- Summary of lessons learned
- Analysis of the procedure
- Analysis of the supplementary activity
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

9. Assessment Rubric

```
1 // Singly Linked List Implementation
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data[50];
7     Node* next;
8 };
9
10 Node* createNode(char* data) {
11     Node* n = new Node;
12     strcpy(n->data, data);
13     n->next = NULL;
14     return n;
15 }
16
17 void insertAtFront(Node* head, char v) {
18     Node* n = new Node(v);
19     if (!head) {
20         head = n;
21         return;
22     }
23     Node* t = head;
24     while (t->next != NULL) {
25         t = t->next;
26     }
27     t->next = n;
28 }
29
30 void insertAtTail(Node* head, char v) {
31     Node* n = new Node(v);
32     head = n;
33 }
34
35 void insertAtIndex(Node* prev, char v) {
36     if (!prev) return;
37     Node* n = new Node(v);
38     prev->next = n;
39 }
40
41 void deleteNode(Node* head, char key) {
42     if (!head) return;
43     if (head->data == key) {
44         Node* temp = head;
45         head = head->next;
46         delete temp;
47     }
48 }
```



```
1 // Linked List Implementation
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data;
7     Node* next;
8 }
9
10 Node* head = NULL;
11 Node* n = NULL;
12
13 void insertAtFront(Node* head, char v) {
14     Node* n = new Node(v);
15     n->next = head;
16     head = n;
17 }
18
19 void insertAtRear(Node* head, char v) {
20     Node* n = new Node(v);
21     if (head == NULL) {
22         head = n;
23         return;
24     }
25     Node* t = head;
26     while (t->next != NULL) {
27         t = t->next;
28     }
29     t->next = n;
30 }
31
32 void insertAtIndex(Node* head, char v, int index) {
33     Node* n = new Node(v);
34     if (index == 0) {
35         insertAtFront(head, v);
36         return;
37     }
38     Node* t = head;
39     for (int i = 1; i < index; i++) {
40         t = t->next;
41     }
42     n->next = t->next;
43     t->next = n;
44 }
45
46 void deleteNode(Node* head, char key) {
47     if (head == NULL) {
48         return;
49     }
50     Node* t = head;
51     if (t->data == key) {
52         head = t->next;
53         delete t;
54     }
55     else {
56         while (t->next != NULL) {
57             if (t->next->data == key) {
58                 t->next = t->next->next;
59                 delete t->next;
60             }
61             t = t->next;
62         }
63     }
64 }
```