

Hands-on Activity 3.1 Linked Lists

LINKED LISTS

Course Code: CPE010

Program: Computer Engineering

Course Title: Data Structures and Algorithms

Date Performed: 9/23/24

Section: CPE21S1

Date Submitted: 9/30/24

Name(s): Aries Rio

Instructor: Ma'am Sayo

6. Output

```

C++ main.cpp x +
c++ main.cpp > f main
1  #include <iostream>
2  using namespace std;
3  class Node {
4      public:
5      char data;
6      Node *next;
7  };
8
9  int main() {
10     // Step 1
11     Node *head = NULL;
12     Node *second = NULL;
13     Node *third = NULL;
14     Node *fourth = NULL;
15     Node *fifth = NULL;
16     Node *last = NULL;
17
18     // Step 2
19     head = new Node;
20     second = new Node;
21     third = new Node;
22     fourth = new Node;
23     fifth = new Node;
24     last = new Node;
25
26     // Step 3
27     head->data = 'C';
28     head->next = second;
29     second->data = 'P';
30     second->next = third;
31     third->data = 'E';
32     third->next = fourth;
33     fourth->data = '0';
34     fourth->next = fifth;
35     fifth->data = '1';
36     fifth->next = last;
37
38     // Step 4
39     last->data = '@';
40     last->next = nullptr;
41
42     // Print the linked list
43     Node *current = head;
44     while (current != nullptr) {
45         cout << current->data << " ";
46         current = current->next;
47     }
48     cout << endl;

```

```

>_ Console x Shell
Run
C P E 0 1 @

```

The code above is a basic implementation of a linked list. Since it doesn't produce any output on its own, I've added some code to display the program's results.

Operation

Screenshot

Transversal

```
1 struct tnode {
2     int data;
3     struct tnode * next;
4 };
5
6 void transversal(struct tnode * head) {
7     struct tnode * temp;
8     temp = head;
9     while (temp != NULL) {
10        printf("%d\t", temp->data);
11        temp = temp->next;
12    }
13    printf("\n");
14 }
15
16 int main() {
17     struct tnode * head = NULL;
18     struct tnode * temp = NULL;
19     int i;
20     for (i = 1; i <= 10; i++) {
21         struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
22         new_node->data = i;
23         new_node->next = NULL;
24         if (head == NULL) {
25             head = new_node;
26         } else {
27             temp->next = new_node;
28         }
29         temp = new_node;
30     }
31     transversal(head);
32     return 0;
33 }
```

```
1 struct tnode {
2     int data;
3     struct tnode * next;
4 };
5
6 void transversal(struct tnode * head) {
7     struct tnode * temp;
8     temp = head;
9     while (temp != NULL) {
10        printf("%d\t", temp->data);
11        temp = temp->next;
12    }
13    printf("\n");
14 }
15
16 int main() {
17     struct tnode * head = NULL;
18     struct tnode * temp = NULL;
19     int i;
20     for (i = 1; i <= 10; i++) {
21         struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
22         new_node->data = i;
23         new_node->next = NULL;
24         if (head == NULL) {
25             head = new_node;
26         } else {
27             temp->next = new_node;
28         }
29         temp = new_node;
30     }
31     transversal(head);
32     return 0;
33 }
```

Insertion at head

```
1 struct tnode {
2     int data;
3     struct tnode * next;
4 };
5
6 void transversal(struct tnode * head) {
7     struct tnode * temp;
8     temp = head;
9     while (temp != NULL) {
10        printf("%d\t", temp->data);
11        temp = temp->next;
12    }
13    printf("\n");
14 }
15
16 void insert_at_head(struct tnode * head, int value) {
17     struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
18     new_node->data = value;
19     new_node->next = head;
20     head = new_node;
21 }
22
23 int main() {
24     struct tnode * head = NULL;
25     struct tnode * temp = NULL;
26     int i;
27     for (i = 1; i <= 10; i++) {
28         struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
29         new_node->data = i;
30         new_node->next = NULL;
31         if (head == NULL) {
32             head = new_node;
33         } else {
34             temp->next = new_node;
35         }
36         temp = new_node;
37     }
38     transversal(head);
39     insert_at_head(head, 11);
40     transversal(head);
41     return 0;
42 }
```

```
1 struct tnode {
2     int data;
3     struct tnode * next;
4 };
5
6 void transversal(struct tnode * head) {
7     struct tnode * temp;
8     temp = head;
9     while (temp != NULL) {
10        printf("%d\t", temp->data);
11        temp = temp->next;
12    }
13    printf("\n");
14 }
15
16 void insert_at_head(struct tnode * head, int value) {
17     struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
18     new_node->data = value;
19     new_node->next = head;
20     head = new_node;
21 }
22
23 int main() {
24     struct tnode * head = NULL;
25     struct tnode * temp = NULL;
26     int i;
27     for (i = 1; i <= 10; i++) {
28         struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
29         new_node->data = i;
30         new_node->next = NULL;
31         if (head == NULL) {
32             head = new_node;
33         } else {
34             temp->next = new_node;
35         }
36         temp = new_node;
37     }
38     transversal(head);
39     insert_at_head(head, 11);
40     transversal(head);
41     return 0;
42 }
```

Insertion at any part of the list

```
1 struct tnode {
2     int data;
3     struct tnode * next;
4 };
5
6 void transversal(struct tnode * head) {
7     struct tnode * temp;
8     temp = head;
9     while (temp != NULL) {
10        printf("%d\t", temp->data);
11        temp = temp->next;
12    }
13    printf("\n");
14 }
15
16 void insert_at_any_part(struct tnode * head, int value, int position) {
17     struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
18     new_node->data = value;
19     new_node->next = NULL;
20     if (position == 1) {
21         insert_at_head(head, value);
22     } else {
23         struct tnode * temp = head;
24         for (int i = 1; i < position; i++) {
25             temp = temp->next;
26         }
27         new_node->next = temp->next;
28         temp->next = new_node;
29     }
30 }
31
32 int main() {
33     struct tnode * head = NULL;
34     struct tnode * temp = NULL;
35     int i;
36     for (i = 1; i <= 10; i++) {
37         struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
38         new_node->data = i;
39         new_node->next = NULL;
40         if (head == NULL) {
41             head = new_node;
42         } else {
43             temp->next = new_node;
44         }
45         temp = new_node;
46     }
47     transversal(head);
48     insert_at_any_part(head, 11, 5);
49     transversal(head);
50     return 0;
51 }
```

```
1 struct tnode {
2     int data;
3     struct tnode * next;
4 };
5
6 void transversal(struct tnode * head) {
7     struct tnode * temp;
8     temp = head;
9     while (temp != NULL) {
10        printf("%d\t", temp->data);
11        temp = temp->next;
12    }
13    printf("\n");
14 }
15
16 void insert_at_any_part(struct tnode * head, int value, int position) {
17     struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
18     new_node->data = value;
19     new_node->next = NULL;
20     if (position == 1) {
21         insert_at_head(head, value);
22     } else {
23         struct tnode * temp = head;
24         for (int i = 1; i < position; i++) {
25             temp = temp->next;
26         }
27         new_node->next = temp->next;
28         temp->next = new_node;
29     }
30 }
31
32 int main() {
33     struct tnode * head = NULL;
34     struct tnode * temp = NULL;
35     int i;
36     for (i = 1; i <= 10; i++) {
37         struct tnode * new_node = (struct tnode *) malloc(sizeof(struct tnode));
38         new_node->data = i;
39         new_node->next = NULL;
40         if (head == NULL) {
41             head = new_node;
42         } else {
43             temp->next = new_node;
44         }
45         temp = new_node;
46     }
47     transversal(head);
48     insert_at_any_part(head, 11, 5);
49     transversal(head);
50     return 0;
51 }
```

Insertion at the end

```
1 // Insertion at the end
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data;
7     Node *next;
8 };
9
10 void insertAtEnd(Node *&head, char value) {
11     if (head == NULL) {
12         head = new Node;
13         head->data = value;
14         head->next = NULL;
15     }
16     else {
17         Node *current = head;
18         while (current->next != NULL) {
19             current = current->next;
20         }
21         current->next = new Node;
22         current->next->data = value;
23         current->next->next = NULL;
24     }
25 }
26
27 void printList(Node *head) {
28     if (head == NULL) {
29         cout << "List is empty." << endl;
30     }
31     else {
32         Node *current = head;
33         while (current != NULL) {
34             cout << current->data << " ";
35             current = current->next;
36         }
37         cout << endl;
38     }
39 }
40
41 int main() {
42     Node *head = NULL;
43     Node *second = NULL;
44     Node *third = NULL;
45     Node *fourth = NULL;
46     Node *fifth = NULL;
47     Node *last = NULL;
48
49     // Step 1
50     head = new Node;
51     head->data = 'C';
52     head->next = NULL;
53
54     // Step 2
55     insertAtEnd(head, 'P');
56
57     // Step 3
58     insertAtEnd(head, 'E');
59
60     // Step 4
61     insertAtEnd(head, 'O');
62
63     // Step 5
64     insertAtEnd(head, 'I');
65
66     printList(head);
67
68     return 0;
69 }
```

Deletion of node

```
1 // Deletion of node
2 #include <iostream>
3 using namespace std;
4
5 struct Node {
6     char data;
7     Node *next;
8 };
9
10 void deleteNode(Node *&head, char value) {
11     if (head == NULL) {
12         cout << "List is empty." << endl;
13         return;
14     }
15     Node *current = head;
16     while (current != NULL) {
17         if (current->data == value) {
18             if (current == head) {
19                 head = current->next;
20             }
21             else {
22                 Node *prev = NULL;
23                 while (prev->next != current) {
24                     prev = prev->next;
25                 }
26                 prev->next = current->next;
27             }
28             delete current;
29             current = current->next;
30         }
31         else {
32             current = current->next;
33         }
34     }
35 }
36
37 void printList(Node *head) {
38     if (head == NULL) {
39         cout << "List is empty." << endl;
40     }
41     else {
42         Node *current = head;
43         while (current != NULL) {
44             cout << current->data << " ";
45             current = current->next;
46         }
47         cout << endl;
48     }
49 }
50
51 int main() {
52     Node *head = NULL;
53     Node *second = NULL;
54     Node *third = NULL;
55     Node *fourth = NULL;
56     Node *fifth = NULL;
57     Node *last = NULL;
58
59     // Step 1
60     head = new Node;
61     head->data = 'C';
62     head->next = NULL;
63
64     // Step 2
65     insertAtEnd(head, 'P');
66
67     // Step 3
68     insertAtEnd(head, 'E');
69
70     // Step 4
71     insertAtEnd(head, 'O');
72
73     // Step 5
74     insertAtEnd(head, 'I');
75
76     printList(head);
77
78     // Step 6
79     deleteNode(head, 'P');
80
81     printList(head);
82
83     return 0;
84 }
```

a

```
1 // main.cpp
2 #include <iostream>
3 using namespace std;
4
5 class Node {
6 public:
7     char data;
8     Node *next;
9 };
10
11 void traverseList(Node *head) {
12     Node *current = head;
13     while (current != NULL) {
14         cout << current->data << " ";
15         current = current->next;
16     }
17 }
18
19 int main() {
20     Node *head = NULL;
21     Node *second = NULL;
22     Node *third = NULL;
23     Node *fourth = NULL;
24     Node *fifth = NULL;
25     Node *last = NULL;
26
27     // Step 1
28     head = new Node;
29     second = new Node;
30     third = new Node;
31     fourth = new Node;
32     fifth = new Node;
33     last = new Node;
34
35     // Step 2
36     head->data = 'C';
37     head->next = second;
38     second->data = 'P';
39     second->next = third;
40     third->data = 'E';
41     third->next = fourth;
42     fourth->data = 'O';
43     fourth->next = fifth;
44     fifth->data = 'I';
45     fifth->next = last;
46     last->data = 'S';
47     last->next = NULL;
48
49     traverseList(head);
50
51     return 0;
52 }
```

b

```
main.cpp x +
main.cpp > f main
1 #include<iostream>
2
3 class Node {
4 public:
5     char data;
6     Node *next;
7 };
8
9 // Function to insert a new node at the head of the list
10 Node* insertAtHead(Node* head, char new_data) {
11     // Step 1: Create a new node
12     Node* new_node = new Node();
13     new_node->data = new_data;
14
15     // Step 2: Make the next of the new node point to the current head
16     new_node->next = head;
17
18     // Step 3: Update the head to be the new node
19     head = new_node;
20
21     return head;
22 }
23
24 // Function to traverse and print the linked list
25 void traverseList(Node* head) {
26     Node* current = head;
27     while (current != nullptr) {
28         std::cout << current->data << " ";
29         current = current->next;
30     }
31     std::cout << std::endl;
32 }
33
34 int main() {
35     // Step 1
36     Node *head = nullptr;
37     Node *second = nullptr;
38
39     // Step 2: Insert nodes at the head
40     head = insertAtHead(head, 'G');
41     head = insertAtHead(head, 'C');
42     head = insertAtHead(head, 'P');
43     head = insertAtHead(head, 'E');
44     head = insertAtHead(head, '1');
45
46     // Step 3: Traverse and print the linked list
47     traverseList(head);
48 }
```

Ln 65, Col 33 • Spaces: 4 History

Linked List after insertion at head: G C P E 1 0

C

```
main.cpp x +
main.cpp > ...
1 #include<iostream>
2
3 class Node {
4 public:
5     char data;
6     Node *next;
7 };
8
9 // Function to insert a new node after a given node
10 void insertAfter(Node* prev_node, char new_data) {
11     if (prev_node == nullptr) {
12         std::cout << "The given previous node cannot be NULL" << std::endl;
13         return;
14     }
15
16     // Step 1: Create a new node
17     Node* new_node = new Node();
18     new_node->data = new_data;
19
20     // Step 2: Make the next of the new node point to the next of prev_node
21     new_node->next = prev_node->next;
22
23     // Step 3: Make the next of prev_node point to the new node
24     prev_node->next = new_node;
25 }
26
27 // Function to insert a new node at the head of the list
28 Node* insertAtHead(Node* head, char new_data) {
29     // Step 1: Create a new node
30     Node* new_node = new Node();
31     new_node->data = new_data;
32
33     // Step 2: Make the next of the new node point to the current head
34     new_node->next = head;
35
36     // Step 3: Update the head to be the new node
37     head = new_node;
38
39     return head;
40 }
```

Ln 65, Col 33 • Spaces: 4 History

Linked List after insertion at head: G C P E 1 0

d

```

1 #include<iostream>
2
3 class Node {
4 public:
5     char data;
6     Node *next;
7 };
8
9 // Function to delete a node with a given key
10 Node* deleteNode(Node* head, char key) {
11     // Store head node
12     Node* temp = head;
13     Node* prev = nullptr;
14
15     // If head node itself holds the key to be deleted
16     if (temp != nullptr && temp->data == key) {
17         head = temp->next; // Changed head
18         delete temp; // Free old head
19         return head;
20     }
21
22     // Search for the key to be deleted, keep track of the previous node
23     while (temp != nullptr && temp->data != key) {
24         prev = temp;
25         temp = temp->next;
26     }
27
28     // If key was not present in linked list
29     if (temp == nullptr) return head;

```

Console Output:

```

Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G P E E 0 1 0

```

e

```

1 #include<iostream>
2
3 class Node {
4 public:
5     char data;
6     Node *next;
7 };
8
9 // Function to delete a node with a given key
10 Node* deleteNode(Node* head, char key) {
11     // Store head node
12     Node* temp = head;
13     Node* prev = nullptr;
14
15     // If head node itself holds the key to be deleted
16     if (temp != nullptr && temp->data == key) {
17         head = temp->next; // Changed head
18         delete temp; // Free old head
19         return head;
20     }
21
22     // Search for the key to be deleted, keep track of the previous node
23     while (temp != nullptr && temp->data != key) {
24         prev = temp;
25         temp = temp->next;
26     }
27
28     // If key was not present in linked list
29     if (temp == nullptr) return head;

```

Console Output:

```

Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G E E 0 1 0

```

f

```

1  #include<iostream>
2
3  class Node {
4  public:
5      char data;
6      Node *next;
7  };
8
9  // Function to delete a node with a given key
10 Node* deleteNode(Node* head, char key) {
11     // Store head node
12     Node* temp = head;
13     Node* prev = nullptr;
14
15     // If head node itself holds the key to be deleted
16     if (temp != nullptr && temp->data == key) {
17         head = temp->next; // Changed head
18         delete temp; // Free old head
19         return head;
20     }
21
22     // Search for the key to be deleted, keep track of the previous

```

Console Output:

```

Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G E E 0 1 0

```

```

1  #include<iostream>
2
3  class Node {
4  public:
5      char data;
6      Node* next;
7      Node* prev;
8  };
9
10 // Function to insert a new node at the head of the list
11 Node* insertAtHead(Node* head, char new_data) {
12     Node* new_node = new Node();
13     new_node->data = new_data;
14     new_node->next = head;
15     new_node->prev = nullptr;
16
17     if (head != nullptr) {
18         head->prev = new_node;
19     }
20
21     head = new_node;
22     return head;

```

Console Output:

```

Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G E E 0 1 0

```

Analysis:

The Node class is defined with three members: data, next, and prev. This allows each node to point to both the next and previous nodes in the list, enabling bidirectional traversal. The insertAtHead function inserts a new node at the beginning of the list. The insertAfter function inserts a new node after a specified node. The deleteNode function deletes a node with a specified key. The traverseList function prints the data of each node in the list. Overall, the code efficiently handles the basic operations of a doubly linked list.

7. Supplementary Activity

Program:

```
114 int main() {
115     Playlist myPlaylist;
116
117     myPlaylist.addSong("Song 1");
118     myPlaylist.addSong("Song 2");
119     myPlaylist.addSong("Song 3");
120
121     std::cout << "Playing all songs in the playlist:" << endl;
122     myPlaylist.playAllSongs();
123
124     std::cout << "\nPlaying next song:" << endl;
125     myPlaylist.nextSong();
126
127     std::cout << "\nPlaying previous song:" << endl;
128     myPlaylist.previousSong();
129
130     std::cout << "\nRemoving 'Song 2' from the playlist." << endl;
131     myPlaylist.removeSong("Song 2");
132
133     std::cout << "\nPlaying all songs in the playlist:" << endl;
134     myPlaylist.playAllSongs();
135
136     return 0;
137 }
```

```
75 // Function to play the next song
76 void nextSong() {
77     if (current) {
78         current = current->next;
79         std::cout << "Playing: " << current->song << endl;
80     }
81 }
82
83 // Function to play the previous song
84 void previousSong() {
85     if (current) {
86         current = current->prev;
87         std::cout << "Playing: " << current->song << endl;
88     }
89 }
90
91 // Function to play all songs in the playlist
92 void playAllSongs() const {
93     if (!head) return;
94
95     Node* temp = head;
96     do {
97         std::cout << temp->song << std::endl;
98         temp = temp->next;
99     } while (temp != head);
100 }
101
102 ~Playlist() {
103     if (!head) return;
104
105     Node* temp = head;
106     do {
107         Node* next = temp->next;
108         delete temp;
109         temp = next;
110     } while (temp != head);
111 }
112 };
```



```

37 // Function to remove a song from the playlist
38 void removeSong(const string& songName) {
39     if (!head) return;
40
41     Node* temp = head;
42
43     // If the song to be deleted is the head
44     if (head->song == songName) {
45         if (head == head->next) {
46             delete head;
47             head = nullptr;
48             current = nullptr;
49         } else {
50             Node* tail = head->prev;
51             head = head->next;
52             tail->next = head;
53             head->prev = tail;
54             delete temp;
55             current = head;
56         }
57         return;
58     }
59
60     // Search for the song to be deleted
61     do {
62         if (temp->song == songName) {
63             temp->prev->next = temp->next;
64             temp->next->prev = temp->prev;
65             if (current == temp) {
66                 current = temp->next;
67             }
68             delete temp;
69             return;
70         }
71         temp = temp->next;
72     } while (temp != head);
73 }

```

```

main.cpp x +
main.cpp > f main Format
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct Node {
6      string song;
7      Node* next;
8      Node* prev; // Adding a previous pointer for doubly linked list
9  };
10
11 class Playlist {
12 private:
13     Node* head;
14     Node* current; // Pointer to keep track of the current song
15
16 public:
17     Playlist() : head(nullptr), current(nullptr) {}
18
19     // Function to add a song to the playlist
20     void addSong(const string& songName) {
21         Node* newNode = new Node();
22         newNode->song = songName;
23         if (!head) {
24             head = newNode;
25             head->next = head;
26             head->prev = head;
27             current = head;
28         } else {
29             Node* tail = head->prev;
30             tail->next = newNode;
31             newNode->prev = tail;
32             newNode->next = head;
33             head->prev = newNode;
34         }
35     }
36

```

Output:

```

>_ Console x Shell +
Show Only Latest Clear History
Run Ask AI 1s on 16:27:15, 09/27 ✓
Playing all songs in the playlist:
Song 1
Song 2
Song 3

Playing next song:
Playing: Song 2

Playing previous song:
Playing: Song 1

Removing 'Song 2' from the playlist.

Playing all songs in the playlist:
Song 1
Song 3

```

8. Conclusion

Mastering linked lists in C++ is a vital step for any aspiring programmer. They provide a practical way to understand dynamic data structures, which are key for efficient data management. Working with linked lists enhances your knowledge of memory management and pointer manipulation—core skills in C++ programming. This foundation prepares you for more advanced topics in computer science, such as trees, graphs, and complex algorithms, while also giving you the practical skills needed to tackle real-world problems. Whether you're optimizing a music playlist, managing dynamic data sets, or implementing a game loop, the concepts you learn from linked lists are directly relevant.