Roman Vasilyev

CISP 430

PROF ROSS

2/15/24

Week 4

Containers

A) Old Programs

Linked List

```
#include "iostream";
using namespace std;

//Our node
struct node {

	node* next;
	char d;
};

//head and tail pointers
node* head = 0;
node* tail = 0;

//function declarations
char remove(void);
void append(char);
int find(char);
void traverse(void);
int isempty(void);

//main for testing the access functions

void main(void)
{



	append('A');
	append('B');
	append('C');
	append('D');
	append('E');
	append('F');
	traverse();

	find('X');
	find('D');
	traverse();

	cout << "Removed" << remove() << endl;
	cout << "Removed" << remove() << endl;
	traverse();

	//empty the list
	cout << "Removed";
```

```
	while (!isempty())
		cout << remove() << "," ;
	cout << endl;
	traverse();

	find('G');
}

void append(char d)
{
	node* p = new node;
	p->next = 0;
	p->d = d;

	if (!(head))
	{
		head = tail = p;
	}
	else
	{
		tail->next = p;
		tail = p;
	}
}

void traverse(void)
{
	node* p = head;
	cout << "The list contains" << endl;

	while (p)
	{
		cout << (char)p->d << " ";
		p = p->next;
	}
	cout << endl;
}

int isempty(void)
{
	if (head)
		return 0;
	else
		return 1;

}

char remove(void)
{
	node* p;
	char temp;
```

```
	if (!head)
		return -1;

	if (head == tail)
	{
		temp = head->d;
		delete head;
		head = tail = 0;
		return temp;
	}
	//more than one node, remove and destroy head node
	p = head;
	head = head->next;
	temp = p->d;
	delete p;
	return temp;
}

int find(char d)
{
	node* c;
	node* pc;

	if (!head)
	{
		cout << d << " not found" << endl;
		return 0;
	}

	if (head == tail)
	{
		if (head->d == d)
		{
			delete head;
			head = tail = 0;
			cout << d << " found" << endl;
			return 1;
		}
		else
		{
			cout << d << " not found" << endl;
			return 0;
		}
	}
	// two or more nodes
	pc = head;
	c = head->next;
	if (pc-> d == d)
	{
		head = head->next;
```

```
                delete pc;
                cout << d << "  found" << endl;
                return 1;
        }
        while (c) {
                if (c->d == d) {
                        pc->next = c->next;
                        if (c == tail)
                                tail = pc;
                        delete c;
                        cout << d <<  "  found" << endl;
                        return 1;
                }
                pc = c;
                c = c->next;
        }
        cout << d << " not found" << endl;

}
```

Microsoft Visual Studio Debug Console

```
The list contains
A B C D E F
X not found
D  found
The list contains
A B C E F
RemovedA
RemovedB
The list contains
C E F
RemovedC,E,F,
The list contains

G not found
```

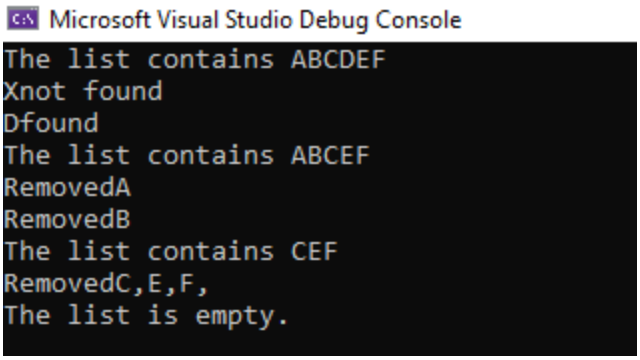
CIRCLE LIST

```
#include "iostream"
using namespace std;

#define SIZE 10
char mylist[SIZE];
int head, tail, used;
```

```
//function declarations
char remove(void);
void append(char);
int find(char);
void traverse(void);
int isempty(void);

//main for testing the access functions

void main(void)
{
	head = tail = used = 0;


	append('A');
	append('B');
	append('C');
	append('D');
	append('E');
	append('F');
	traverse();

	find('X');
	find('D');
	traverse();

	cout << "Removed" << remove() << endl;
	cout << "Removed" << remove() << endl;
	traverse();

	//empty the list
	cout << "Removed";
	while (!isempty())
		cout << remove() << ",";
	cout << endl;
	traverse();

	find('G');
}

void append(char d)
{
	if (!used) {
		mylist[tail] = d;
		used++;
		return;
	}
```

```
	if ((tail + 1) % SIZE == head) {
		cout << " Overflow. Element not appended. \n";
		return;
	}
	tail = (tail + 1) % SIZE;
	mylist[tail] = d;
	used++;
}

void traverse(void)
{
	char p;
	if (isempty()) {
		cout << "The list is empty. \n";
		return;
	}
	if (used == 1) {
		cout << "The list continues " << mylist[head] << endl;
		return;
	}
	p = head;
	printf("The list contains ");
	do {
		printf("%c", mylist[p]);
		p = (p + 1) % SIZE;
	} while (p != (tail + 1) % SIZE);
	cout << endl;
}

int isempty(void)
{
	if (used)
		return 0;
	else
		return 1;

}

char remove(void)
{

	char temp;

	if (isempty()) {
		return -1;
	}


	if (used == 1)
	{
		used = 0;
```

```
                return mylist[head];
        }
        temp = mylist[head];
        head = (head + 1) % SIZE;
        used--;
        return temp;
}


int find(char d)
{
        int p;

        if (isempty()) {
                return 0;

        }
        if (used == 1) {
                if (mylist[head] == d) {
                        used = 0;
                        cout << d << "found" << endl;
                        return -1;
                }
                else {
                        cout << d << "not found " << endl;
                        return 0;
                }
        }

        p = head;
        do {
                if (mylist[p] == d) {
                        while (p != tail) {
                                mylist[p] = mylist[(p + 1) % SIZE];
                                p = (p + 1) % SIZE;
                        }
                        tail--;
                        if (tail < 0) tail = SIZE - 1;
                        used--;
                        cout << d << "found" << endl;
                        return 1;
                }
                p = (p + 1) % SIZE;
        } while (p != (tail + 1) % SIZE);
        cout << d << "not found" << endl;
        return 0;

}
```



CIRCLE:

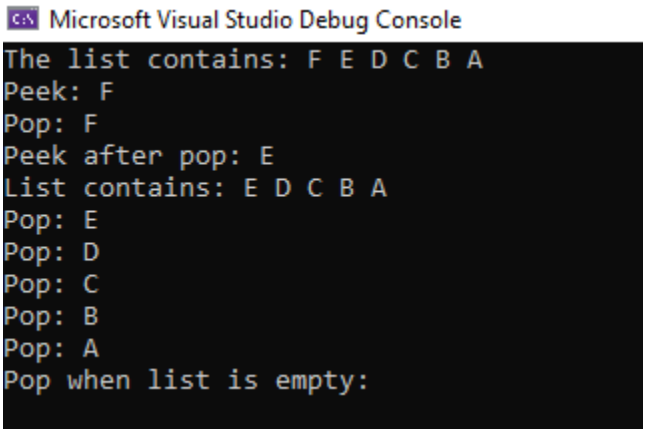
1. Stack

```
#include <iostream>

using namespace std;


#define SIZE 10


char mylist[SIZE];

int head, tail, used;


// Function declarations

void push(char);

char pop(void);

char peek(void);

bool isempty(void);

void display(void);


// Main for testing the stack functions

int main() {

   head = tail = used = 0;
```

```
    push('A');
    push('B');
    push('C');
    push('D');
    push('E');
    push('F');
    cout << "The list contains: ";
    display();

    cout << "Peek: " << peek() << endl;
    cout << "Pop: " << pop() << endl;
    cout << "Peek after pop: " << peek() << endl;

    cout << "List contains: ";
    display();

    while (!isempty()) {
        cout << "Pop: " << pop() << endl;
    }

    cout << "Pop when list is empty: " << pop() << endl;

    return 0;
}

void push(char data) {
    if ((tail + 1) % SIZE == head) {
        cout << "Stack Overflow. Element not pushed.\n";
        return;
```

```
    }
    head = (head - 1 + SIZE) % SIZE;
    mylist[head] = data;
    used++;
}

char pop(void) {
    if (isempty()) {
        return -1;
    }
    char temp = mylist[head];
    head = (head + 1) % SIZE;
    used--;
    return temp;
}

char peek(void) {
    if (isempty()) {
        return -1;
    }
    return mylist[head];
}

bool isempty(void) {
    return used == 0;
}

void display(void) {
    if (isempty()) {
```

```
        cout << "List is empty." << endl;
        return;
    }
    int i = head;
    do {
        cout << mylist[i] << " ";
        i = (i + 1) % SIZE;
    } while (i != tail);
    cout << mylist[i] << endl;
}
```



Summary. push, pop, peek, and isempty all have a time complexity of O(1) because they involve a constant number of operations regardless of the size of the circular list (stack).

display has a time complexity of O(N) because it iterates through each element of the circular list, which could have at most SIZE elements.

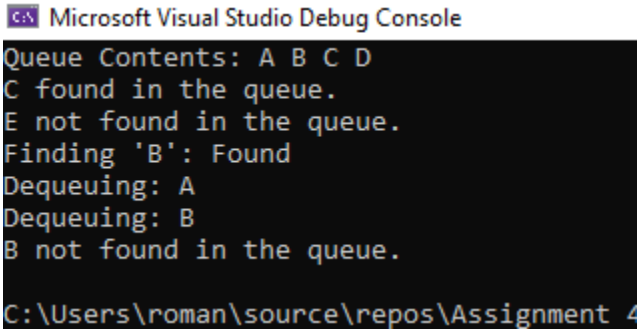
QUEUE

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

#define SIZE 10
```

```
char mylist[SIZE];
int head, tail, used;

// Function declarations
void q(char);
char dq(void);
bool isempty(void);
void traverse(char);
void display(void);
int find(char);

// Main for testing the queue functions
int main() {
   head = tail = used = 0;

   q('A');
   q('B');
   q('C');
   q('D');

   display();
   traverse('C');
   traverse('E');

   cout << "Finding 'B': " << (find('B') ? "Found" : "Not Found") << endl;

   cout << "Dequeuing: " << dq() << endl;
   cout << "Dequeuing: " << dq() << endl;

   traverse('B');

   return 0;
}

void q(char data) {
   if ((tail + 1) % SIZE == head) {
      cout << "Queue Overflow. Element not appended.\n";
      return;
   }
   mylist[tail] = data;
   tail = (tail + 1) % SIZE;
   used++;
}
```

```
char dq(void) {
   if (isempty()) {
      return -1;
   }
   char temp = mylist[head];
   head = (head + 1) % SIZE;
   used--;
   return temp;
}

bool isempty(void) {
   return used == 0;
}

void traverse(char key) {
   bool found = false;
   int i = head;
   while (i != tail) {
      if (mylist[i] == key) {
         found = true;
         break;
      }
      i = (i + 1) % SIZE;
   }
   if (mylist[i] == key) {
      found = true;
   }
   if (found) {
      cout << key << " found in the queue." << endl;
   } else {
      cout << key << " not found in the queue." << endl;
   }
}

void display(void) {
   if (isempty()) {
      cout << "Queue is empty." << endl;
      return;
   }
   cout << "Queue Contents: ";
   int i = head;
   while (i != tail) {
```

```
        cout << mylist[i] << " ";
        i = (i + 1) % SIZE;
    }
    cout << mylist[i] << endl;
}

int find(char key) {
    bool found = false;
    int i = head;
    while (i != tail) {
        if (mylist[i] == key) {
            found = true;
            break;
        }
        i = (i + 1) % SIZE;
    }
    if (mylist[i] == key) {
        found = true;
    }
    return found;
}
```



q, dq, and isempty all have a time complexity of O(1) because they involve a constant number of operations regardless of the size of the circular list (queue).
traverse, display, and find have a time complexity of O(N) because they may need to iterate through all elements of the circular list (queue), which can be at most SIZE elements

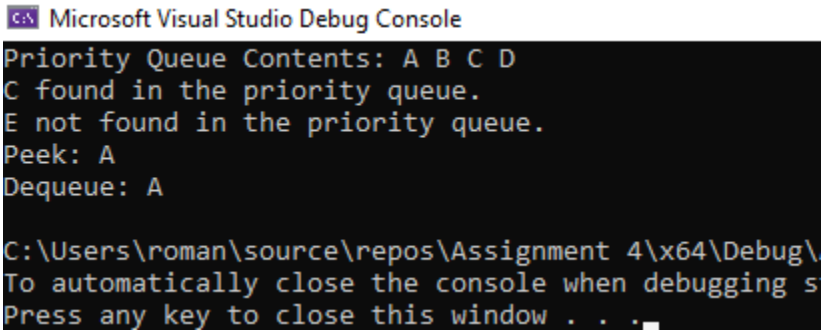
Priority Queue

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

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

Node* head = nullptr;

// Function declarations
void insert(char);
char dq(void);
char peek(void);
bool isEmpty(void);
void display(void);
void traverse(char);

// Main for testing the priority queue functions
int main() {
   insert('C');
   insert('A');
   insert('D');
   insert('B');

   display();
   traverse('C');
   traverse('E');

   cout << "Peek: " << peek() << endl;
   cout << "Dequeue: " << dq() << endl;

   return 0;
}

void insert(char data) {
   Node* newNode = new Node;
   newNode->data = data;
   newNode->next = nullptr;

   if (!head || head->data >= data) {
      newNode->next = head;
      head = newNode;
   }
   else {
```

```
        Node* current = head;
        while (current->next && current->next->data < data) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}

char dq(void) {
    if (isEmpty()) {
        return -1;
    }
    char data = head->data;
    Node* temp = head;
    head = head->next;
    delete temp;
    return data;
}

char peek(void) {
    if (isEmpty()) {
        return -1;
    }
    return head->data;
}

bool isEmpty(void) {
    return !head;
}

void display(void) {
    if (isEmpty()) {
        cout << "Priority Queue is empty." << endl;
        return;
    }
    cout << "Priority Queue Contents: ";
    Node* current = head;
    while (current) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
```

```
}

void traverse(char key) {
    bool found = false;
    Node* current = head;
    while (current) {
        if (current->data == key) {
            found = true;
            break;
        }
        current = current->next;
    }
    if (found) {
        cout << key << " found in the priority queue." << endl;
    }
    else {
        cout << key << " not found in the priority queue." << endl;
    }
}
```



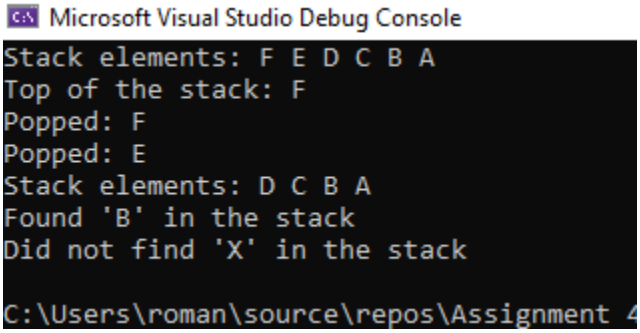
insert, display, and traverse have a time complexity of O(n) because they may need to traverse the entire priority queue, where n is the number of elements in the priority queue.
dq, peek, and isEmpty have a time complexity of O(1) because they involve constant-time operations regardless of the size of the priority queue.

LINKED LIST

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

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

class Stack {
private:
   node* head;

public:
   Stack() : head(nullptr) {}

   void push(char data) {
      node* newNode = new node;
      newNode->data = data;
      newNode->next = head;
      head = newNode;
   }

   char pop() {
      if (isempty())
         return -1;

      char data = head->data;
      node* temp = head;
      head = head->next;
      delete temp;
      return data;
   }

   char peek() {
      if (isempty())
         return -1;

      return head->data;
```

```
    }

    bool isempty() {
        return head == nullptr;
    }

    void traverse() {
        node* current = head;
        cout << "Stack elements: ";
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }

    bool find(char data) {
        node* current = head;
        while (current != nullptr) {
            if (current->data == data)
                return true;
            current = current->next;
        }
        return false;
    }
};

int main() {
    Stack stack;
    stack.push('A');
    stack.push('B');
    stack.push('C');
    stack.push('D');
    stack.push('E');
    stack.push('F');
    stack.traverse();

    cout << "Top of the stack: " << stack.peek() << endl;

    cout << "Popped: " << stack.pop() << endl;
    cout << "Popped: " << stack.pop() << endl;

    stack.traverse();
```

```
    if (stack.find('B'))
        cout << "Found 'B' in the stack" << endl;

    if (stack.find('X'))
        cout << "Found 'X' in the stack\n";
    else
        cout << "Did not find 'X' in the stack\n";

    return 0;
}
```



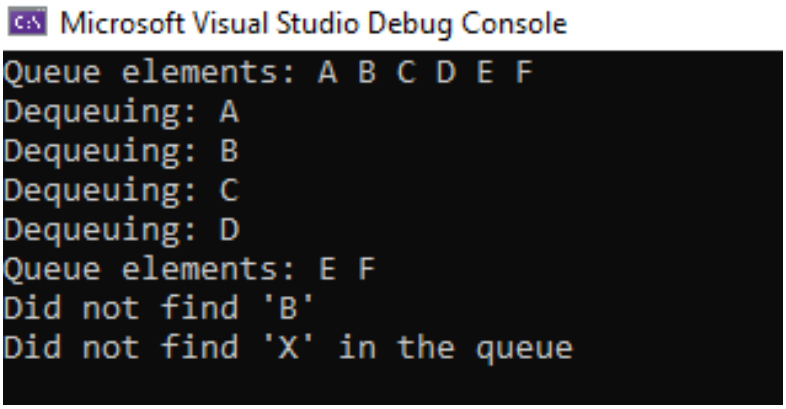
Push, Pop, Peek, and IsEmpty operations have a time complexity of O(1) because they involve constant-time operations that do not depend on the size of the stack. * Traverse and Find operations have a time complexity of O(n) because they involve traversing the entire stack, and the time taken grows linearly with the number of elements in the stack.

STACKED LINKED

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

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

class Queue {
private:
    node* head;
    node* tail;

public:
    Queue() : head(nullptr), tail(nullptr) {}
```

```
void q(char data) {
    node* newNode = new node;
    newNode->data = data;
    newNode->next = nullptr;
    if (isempty()) {
        head = tail = newNode;
    }
    else {
        tail->next = newNode;
        tail = newNode;
    }
}

char dq() {
    if (isempty())
        return -1;

    char data = head->data;
    node* temp = head;
    head = head->next;
    if (head == nullptr)
        tail = nullptr; // Reset tail if queue becomes empty
    delete temp;
    return data;
}

bool isempty() {
    return head == nullptr;
}

void traverse() {
    node* current = head;
    cout << "Queue elements: ";
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

bool find(char data) {
    node* current = head;
```

```
        while (current != nullptr) {
            if (current->data == data)
                return true;
            current = current->next;
        }
        return false;
    }
};

int main() {
    Queue queue;
    queue.q('A');
    queue.q('B');
    queue.q('C');
    queue.q('D');
    queue.q('E');
    queue.q('F');

    queue.traverse();

    cout << "Dequeuing: " << queue.dq() << endl;
    cout << "Dequeuing: " << queue.dq() << endl;
    cout << "Dequeuing: " << queue.dq() << endl;
    cout << "Dequeuing: " << queue.dq() << endl;
    queue.traverse();

    if (queue.find('B'))
        cout << "Found 'B' in the queue\n";
    else cout << "Did not find 'B' " << endl;

    if (queue.find('X'))
        cout << "Found 'C' in the queue\n";
    else
        cout << "Did not find 'X' in the queue\n";

    return 0;
}
```



Queue, Dequeue, and IsEmpty operations have a time complexity of O(1) because they involve constant-time operations that do not depend on the size of the queue.
Traverse and Find operations have a time complexity of O(n) because they involve traversing the entire queue, and the time taken grows linearly with the number of elements in the queue.
PRIORITY QUEUE

```
#include <iostream>

using namespace std;


// Our node

struct Node {

    Node* next;

    char data;

};


// Head and tail pointers

Node* head = nullptr;

Node* tail = nullptr;


// Function declarations

void insert(char data);

char dq();

char peek();

bool isEmpty();
```

```
void traverse();
int find(char data);

int main() {
    insert('A');
    insert('B');
    insert('C');
    insert('D');
    insert('E');
    insert('F');

    traverse();

    cout << "Peek: " << peek() << endl;
    cout << "Dequeue: " << dq() << endl;
    traverse();

    while (!isEmpty()) {
        cout << "Dequeued: " << dq() << endl;
    }

    if (find('B'))
        cout << "Found 'B' " << endl;
    else cout << "Did not find 'B' " << endl;

    if (find('X'))
        cout << "Did not find 'X' " << endl;

    return 0;
}
```

```
void insert(char data) {
    Node* newNode = new Node;
    newNode->next = nullptr;
    newNode->data = data;

    if (head == nullptr) {
        head = tail = newNode;
    }
    else {
        if (data <= head->data) { // Insert at head
            newNode->next = head;
            head = newNode;
        }
        else if (data >= tail->data) { // Insert at tail
            tail->next = newNode;
            tail = newNode;
        }
        else { // Insert in between
            Node* current = head;
            while (current->next != nullptr && current->next->data < data) {
                current = current->next;
            }
            newNode->next = current->next;
            current->next = newNode;
        }
    }
}

char dq() {
```

```
    if (isEmpty())
        return -1;

    Node* temp = head;
    char data = temp->data;
    head = head->next;
    delete temp;
    if (head == nullptr)
        tail = nullptr;
    return data;
}

char peek() {
    if (isEmpty())
        return -1;
    return head->data;
}

bool isEmpty() {
    return head == nullptr;
}

void traverse() {
    Node* current = head;
    cout << "Priority Queue: ";
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
```

```
}

int find(char data) {
    Node* current = head;
    int index = 0;
    while (current != nullptr) {
        if (current->data == data)
            return index;
        current = current->next;
        index++;
    }
    return -1;
}
```

Microsoft Visual Studio Debug Console

```
Priority Queue: A B C D E F
Peek: A
Dequeue: A
Priority Queue: B C D E F
Dequeued: B
Dequeued: C
Dequeued: D
Dequeued: E
Dequeued: F
Found 'B'
Did not find 'X'
```

the time complexity for insertion and finding elements in the middle of the list is O(n), whereas the time complexity for other operations is O(1).

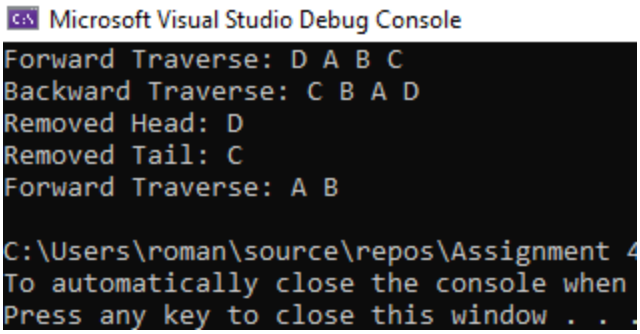
DOUBLE LINKED LIST

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

// Node structure for Double Linked List
struct Node {
```

```
    char data;
    Node* next;
    Node* prev;
};

// Head and tail pointers
Node* head = nullptr;
Node* tail = nullptr;

// Function declarations
void appendTail(char data);
void appendHead(char data);
char removeTail();
char removeHead();
void traverseFWD();
void traverseBWD();
bool isEmpty();

int main() {
    appendTail('A');
    appendTail('B');
    appendTail('C');
    appendHead('D');
    traverseFWD();
    traverseBWD();

    cout << "Removed Head: " << removeHead() << endl;
    cout << "Removed Tail: " << removeTail() << endl;
    traverseFWD();

    return 0;
}

void appendTail(char data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = nullptr;

    if (isEmpty()) {
        head = tail = newNode;
        newNode->prev = nullptr;
    }
    else {
```

```
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
}

void appendHead(char data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->prev = nullptr;

    if (isEmpty()) {
        head = tail = newNode;
        newNode->next = nullptr;
    }
    else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}

char removeTail() {
    if (isEmpty())
        return -1;

    Node* temp = tail;
    char data = temp->data;

    if (head == tail) { // Only one node
        delete temp;
        head = tail = nullptr;
    }
    else {
        tail = tail->prev;
        tail->next = nullptr;
        delete temp;
    }
    return data;
}

char removeHead() {
    if (isEmpty())
```

```
        return -1;

    Node* temp = head;
    char data = temp->data;

    if (head == tail) { // Only one node
        delete temp;
        head = tail = nullptr;
    }
    else {
        head = head->next;
        head->prev = nullptr;
        delete temp;
    }
    return data;
}

void traverseFWD() {
    Node* current = head;
    cout << "Forward Traverse: ";
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

void traverseBWD() {
    Node* current = tail;
    cout << "Backward Traverse: ";
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->prev;
    }
    cout << endl;
}

bool isEmpty() {
    return head == nullptr;
}
```



Append to Tail (appendTail):
Regardless of the size of the list, the time complexity is O(1) because we always have a pointer to the tail node, so we can directly append to it.
Append to Head (appendHead):
the time complexity is O(1) because we always have a pointer to the head node, allowing us to directly prepend to it.
Remove Tail (removeTail):
Regardless of the size of the list, the time complexity is O(1) because we always have a pointer to the tail node, so we can directly remove it.
Remove Head (removeHead):
the time complexity is O(1) because we always have a pointer to the head node, allowing us to directly remove it.
Traverse Forward (traverseFWD):
The time complexity is O(n), where n is the number of elements in the list, because we need to visit each node once to print its data.
Traverse Backward (traverseBWD):
the time complexity is O(n) because we need to visit each node once to print its data.
Is Empty (isEmpty)
The time complexity is O(1) because it only involves checking if the head pointer is null.