Simply linked list

```
template <class T>
class List
{
public:
	List() = default;
	~List();
	List(const List&);
	List& operator=(const List&);
	List(List&&);
	List& operator=(List&&);
	List(const std::initializer_list<T>&);
	bool isEmpty()const;
	void display()const;
	void push_back(const T&);
	void push_front(const T&);
	void pop_back();
	void pop_front();
	T max()const;
	void search(const T&);
	void insertNodeAfterKey(const T&, const T&);
	void deleteNodeWithKey(const T&);
	bool isSorted()const;
	void removeDuplicates();
	void reverse();
	List& concatenate(List&);
	List& merge(List&);
	bool isLoop()const;
	T middle()const;
	T intersectionPoint(const List&)const;
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& data) :data{ data }, next{ nullptr }{}
		T data {};
		std::shared_ptr<Node> next{ nullptr };
	};
	std::shared_ptr<Node<T>> first{ nullptr };
	std::shared_ptr<Node<T>> last{ nullptr };
};
template <class T>
List<T>::~List()
{
	while (first != nullptr)
	{
		first = first->next;
	}
}
template<class T>
List<T>::List(const List& other)
{
	auto current = other.first;
	while (current != nullptr)
	{
		push_back(current->data);
		current = current->next;
	}
}
template<class T>
```

```
List<T>& List<T>::operator=(const List& other)
{
	if (this != &other)
	{
		while (!isEmpty())
		{
			pop_back();
		}
		auto current = other.first;
		while (current != nullptr)
		{
			push_back(current->data);
			current = current->next;
		}
	}
	return *this;
}
template<class T>
List<T>::List(List&& other) :first{ std::move(other.first) }, last{ std::move(other.last) }
{
	other.first = nullptr;
	other.last = nullptr;
}
template<class T>
List<T>& List<T>::operator=(List&& other)
{
	if (this != &other)
	{
		while (!isEmpty())
		{
			pop_back();
		}
		first = std::move(other.first);
		last = std::move(other.last);
		other.first = nullptr;
		other.last = nullptr;
	}
	return *this;
}
template<class T>
List<T>::List(const std::initializer_list<T>& il)
{
	for (auto p = il.begin();p != il.end();p++)
	{
		push_back(*p);
	}
}
template <class T>
bool List<T>::isEmpty()const
{
	return first == nullptr && last == nullptr;
}
template <class T>
void List<T>::display()const
{
	if (isEmpty())
	{
		std::cout << "There is no list to display!\n";
		return;
	}
	auto current = first;
	std::cout << "List is:";
```

```
	while (current != nullptr)
	{
		std::cout << current->data << ' ';
		current = current->next;
	}
	std::cout << '\n';
}
template<class T>
void List<T>::push_back(const T& newData)
{
	if (isEmpty())
	{
		first = std::make_shared<Node<T>>(newData);
		last = first;
		return;
	}
	last->next = std::make_shared<Node<T>>(newData);
	last = last->next;
}
template<class T>
void List<T>::push_front(const T& newData)
{
	if (isEmpty())
	{
		first = std::make_shared<Node<T>>(newData);
		last = first;
		return;
	}
	auto newNode = std::make_shared<Node<T>>(newData);
	newNode->next = first;
	first = newNode;
}
template<class T>
void List<T>::pop_back()
{
	if (isEmpty())
	{
		std::cout << "Cannot pop the back of an empty list!\n";
		return;
	}
	if (first == last)
	{
		first = last = nullptr;
		return;
	}
	auto current = first;
	for ( ; current->next->next ;current = current->next);
	last = current;
	last->next = nullptr;
}
template<class T>
void List<T>::pop_front()
{
	if (isEmpty())
	{
		std::cout << "Cannot pop the front of an empty list!\n";
		return;
	}
	if (first == last)
	{
		first = last = nullptr;
		return;
```

```
	}
	first = first->next;
}
template<class T>
T List<T>::max() const
{
	if (isEmpty())
	{
		std::cout << "Cannot found the maximum of an empty list!\n";
		return T(); //better to throw
	}
	T Max = first->data;
	auto current = first;
	while (current != nullptr)
	{
		if (current->data > Max)
		{
			Max = current->data;

		}
		current = current->next;
	}
	return Max; //v. locala, e mutata
}
template<class T>
void List<T>::search(const T& key)//improved with move to front
{
	if (isEmpty())
	{
		std::cout << "Cannot search in an empty list!\n";
		return;
	}
	int position = 1;
	if (first->data == key)
	{
		std::cout << "The node with key " << key << " is at position " << position <<
                                                                        '\n';
		return;
	}
	for (auto current = first; current->next ; current = current->next)
	{
		position++;
		if (current->next->data == key)
		{
			std::cout << "The node with key " << key << " is at position " <<
                                                                    position << '\n';
			auto temp = current->next;
			current->next = current->next->next;
			temp->next = first;
			first = temp;
			return;
		}
		if (current->next == last)
		{
			std::cout << "The node with key " << key << " is not in the list!\n";
			return;
		}
	}
}
template<class T>
void List<T>::insertNodeAfterKey(const T& key, const T& newData)
{
```

```
	if (isEmpty())
	{
		std::cout << "List is empty.No key found!\n";
		return;
	}
	auto current = first;
	while (current != nullptr)
	{
		if (current->data == key)
		{
			auto newNode = std::make_shared<Node<T>>(newData);
			if (current == last)
			{
				last->next = newNode;
				last = newNode;
				return;
			}
			newNode->next = current->next;
			current->next = newNode;
			return;
		}
		if (current == last)
		{
			std::cout << "The node with key " << key << " is not in the list!\n";
			return;
		}
		current = current->next;
	}
}
template<class T>
void List<T>::deleteNodeWithKey(const T& key)
{
	if (isEmpty())
	{
		std::cout << "List is empty.Cannot delete the node with key " << key << '\n';
		return;
	}
	if (first->data == key)
	{
		if (first == last)
		{
			first = last = nullptr;
			return;
		}
		first = first->next;
		return;
	}
	for (auto current = first; current->next ;current = current->next)
	{
		if (current->next->data == key)
		{
			if (current->next == last)
			{
				last = current;
			}
			current->next = current->next->next;
			return;
		}
		if (current->next == last)
		{
			std::cout << "Cannot delete the node with key " << key << ".Is not in
the list.\n";
```

```
                        return;
                }
        }
}
template<class T>
bool List<T>::isSorted() const
{
        if (isEmpty())
        {
                std::cout << "List is empty.Cannot tell if is sorted!\n";
                return false;
        }
        auto temp = first->data;
        auto current = first->next;
        while (current != nullptr)
        {
                if (current->data < temp)
                {
                        return false;
                }
                temp = current->data;
                current = current->next;
        }
        return true;
}
template<class T>
//consecutive duplicates
void List<T>::removeDuplicates()
{
        if (isEmpty())
        {
                std::cout << "List is empty.Cannot remove duplicates!\n";
                return;
        }
        auto current = first;
        while (current->next != nullptr)
        {
                if (current->data != current->next->data)
                {
                        current = current->next;
                }
                else
                {
                        current->next = current->next->next;
                }
        }
}
template<class T>
//using sliding pointers(needs 3 pointers), reversing only links not data
void List<T>::reverse()
{
        if (isEmpty())
        {
                std::cout << "Cannot reverse an empty list!\n";
                return;
        }
        last = first;
        std::shared_ptr<Node<T>> q{ nullptr }, r{ nullptr };
        auto p = first;
        while (p != nullptr)
        {
                r = q;
```

```
			q = p;
			p = p->next;
			q->next = r;
		}
		first = q;
}
template<class T>
List<T>& List<T>::concatenate(List& other)
{
		last->next = other.first;
		last = other.last;
		return *this;
}
template<class T>
//both lists should be sorted
List<T>& List<T>::merge(List& other)
{
		if (first->data > other.first->data)
		{
			auto temp = this->first;
			this->first = other.first;
			other.first = temp;
		}
		auto current = first;
		auto currentOther = other.first;
		if (first->data < other.first->data)
		{
			last = first;
			current = current->next;
			last->next = nullptr;
		}
		else
		{
			last = other.first;
			currentOther = currentOther->next;
			last->next = nullptr;
		}
		while(current != nullptr && currentOther != nullptr)
		{
			if (current->data < currentOther->data)
			{
				last->next = current;
				last = current;
				current = current->next;
				last->next = nullptr;
			}
			else
			{
				last->next = currentOther;
				last = currentOther;
				currentOther = currentOther->next;
				last->next = nullptr;
			}
		}
		if (current != nullptr)
		{
			last->next = current;
		}
		if (currentOther != nullptr)
		{
			last->next = currentOther;
		}
```

```
	return *this;
}
template<class T>
bool List<T>::isLoop() const
{
	std::shared_ptr<Node<T>> p, q;
	p = q = first;
	do
	{
		p = p->next;// 1 step
		q = q->next;
		q = q != nullptr ? q->next : q;//2 steps
	} while (p && q && p != q);
	if (p == q)
	{
		return true;
	}
	else
	{
		return false;//is linear
	}
}
template<class T>
//in a single scan
T List<T>::middle() const
{
	auto p = first;
	auto q = first;
	while (q != nullptr)
	{
		q = q->next;
		if (q != nullptr)
		{
			q = q->next;//moves to steps
		}
		if (q !=nullptr)
		{
			p = p->next;//moves 1 step
		}
	}
	return p->data;
}
template<class T>
T List<T>::intersectionPoint(const List& other) const
{
	std::stack<std::shared_ptr<Node<T>>> s1, s2;
	auto p = first;
	while (p != nullptr)
	{
		s1.push(p);
		p = p->next;
	}
	p = other.first;
	while (p != nullptr)
	{
		s2.push(p);
		p = p->next;
	}
	std::shared_ptr<Node<T>> ip{ nullptr };
	while (s1.top() == s2.top())
	{
		ip = s1.top();
```

```
			s1.pop();
			s2.pop();
		}
		if (ip != nullptr)
		{
			return ip->data;
		}
		else
		{
			std::cout << "There is no intersection point\n";
			return T();
		}
}
```

Circular simply linked list

```
template <class T>
class List
{
public:
		List() = default;
		~List();
		List(const std::initializer_list<T>&);
		bool isEmpty()const;
		void display()const;
		void push_back(const T&);
		void reverse();
private:
		template <class T>
		struct Node
		{
			Node() = default;
			explicit Node(const T& data) :data{ data }, next{ nullptr }{}
			T data{};
			std::shared_ptr<Node> next{ nullptr };
		};
		std::shared_ptr<Node<T>> first{ nullptr };
};
template <class T>
List<T>::~List()
{
		if (isEmpty())
		{
			return;
		}
		auto temp = first;
		do
		{
			first = first->next;
		} while (first != temp);
}
template<class T>
bool List<T>::isEmpty() const
{
		return first == nullptr;
}
template<class T>
void List<T>::display() const
{
		if (isEmpty())
		{
			std::cout << "There is no list to display!\n";
```

```
		return;
	}
	std::cout << "List is:";
	auto current = first;
	do
	{
		std::cout << current->data << ' ';
		current = current->next;
	} while (current != first);
	std::cout << '\n';
}
template<class T>
void List<T>::push_back(const T& newData)
{
	if (isEmpty())
	{
		first = std::make_shared<Node<T>>(newData);
		first->next = first;
		return;
	}
	auto p = first;
	while(p->next != first)
	{
		p = p->next;
	}
	auto newNode = std::make_shared<Node<T>>(newData);
	p->next = newNode;
	newNode->next = first;
}
template<class T>
List<T>::List(const std::initializer_list<T>& il)
{
	for (auto p = il.begin();p != il.end();p++)
	{
		push_back(*p);
	}
}
template<class T>
void List<T>::reverse()
{
	if (isEmpty())
	{
		std::cout << "Cannot reverse an empty list!\n";
		return;
	}
	std::shared_ptr<Node<T>> q{ nullptr }, r{ nullptr };
	auto p = first;
	do
	{
		r = q;
		q = p;
		p = p->next;
		q->next = r;
	}while (p != first);
	p->next = q;
	first = q;
}
```

Doubly linked list

```
template <class T>
class List
```

```
{
public:
	List() = default;
	List(const List&)=default;//posibil shallow copy
	List& operator=(const List&)=default;//posibil shallow copy
	List(List&&)=default;//tb. default sau implementat
	List& operator=(List&&)=default;//tb. default
	~List();
	List(const std::initializer_list<T>&);
	bool isEmpty()const;
	void displayForward()const;
	void displayBackward()const;
	void push_back(const T&);
	void push_front(const T&);
	void pop_back();
	void pop_front();
	void insertNodeAfterKey(const T&, const T&);
	void deleteNodeWithKey(const T&);
	void reverse();
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& data) :data{ data }, next{}, prev{}{}
		T data{};
		std::shared_ptr<Node> next{ nullptr };
		std::weak_ptr<Node> prev{ nullptr };
	};
	std::shared_ptr<Node<T>> first{ nullptr };
	std::shared_ptr<Node<T>> last{ nullptr };
};
template<class T>
List<T>::~List()
{
	while (first != nullptr)
	{
		first = first->next;
	}
}
template<class T>
List<T>::List(const std::initializer_list<T>& il)
{
	for (auto p = il.begin(); p != il.end() ; ++p)
	{
		push_back(*p);
	}
}
template <class T>
bool List<T>::isEmpty()const
{
	return first == nullptr && last == nullptr;
}
template<class T>
void List<T>::displayForward() const
{
	if (isEmpty())
	{
		std::cout << "List is empty.Cannot display forward\n";
		return;
	}
	auto current = first;
```

```
	std::cout << "List is:";
	while (current != nullptr)
	{
		std::cout << current->data << ' ';
		current = current->next;
	}
	std::cout << '\n';
}
template<class T>
void List<T>::displayBackward() const
{
	if (isEmpty())
	{
		std::cout << "List is empty.Cannot display backward\n";
		return;
	}
	auto current = last;
	std::cout << "List is:";
	while (current != nullptr)
	{
		std::cout << current->data << ' ';
		current = current->prev.lock();
	}
	std::cout << '\n';
}
template<class T>
void List<T>::push_back(const T& newData)
{
	if (isEmpty())
	{
		first = std::make_shared<Node<T>>(newData);
		last = first;
		return;
	}
	auto newNode = std::make_shared<Node<T>>(newData);
	newNode->prev = last;
	last->next = newNode;
	last = newNode;
}
template<class T>
void List<T>::push_front(const T& newData)
{
	if (isEmpty())
	{
		first = std::make_shared<Node<T>>(newData);
		last = first;
		return;
	}
	auto newNode = std::make_shared<Node<T>>(newData);
	newNode->next = first;
	first->prev = newNode;
	first = newNode;
}
template<class T>
void List<T>::pop_back()
{
	if (isEmpty())
	{
		std::cout << "Cannot pop the back of an empty list\n";
		return;
	}
	if (first == last)
```

```
	{
		first = last = nullptr;
		return;
	}
	last = last->prev.lock();
	last->next = nullptr;
}
template<class T>
void List<T>::pop_front()
{
	if (isEmpty())
	{
		std::cout << "Cannot pop the front of an empty list\n";
		return;
	}
	if (first == last)
	{
		first = last = nullptr;
		return;
	}
	first = first->next;
	first->prev.lock() = nullptr;
}
template<class T>
void List<T>::insertNodeAfterKey(const T& key, const T& newData)
{
	if (isEmpty())
	{
		std::cout << "List is empty.No key found!\n";
		return;
	}
	auto current = first;
	while (current != nullptr)
	{
		if (current->data == key)
		{
			auto newNode = std::make_shared<Node<T>>(newData);
			if (current == last)
			{
				last->next = newNode;
				newNode->prev = last;
				last = newNode;
				return;
			}
			newNode->next = current->next;
			current->next->prev = newNode;
			current->next = newNode;
			newNode->prev = current;
			return;
		}
		if (current == last)
		{
			std::cout << "The node with key " << key << " is not in the list!\n";
			return;
		}
		current = current->next;
	}
}
template<class T>
void List<T>::deleteNodeWithKey(const T& key)
{
	if (isEmpty())
```

```
	{
		std::cout << "List is empty.Cannot delete the node with key " << key << '\n';
		return;
	}
	if (first->data == key)
	{
		if (first == last)
		{
			first = last = nullptr;
			return;
		}
		first = first->next;
		first->prev.lock() = nullptr;
		return;
	}
	for (auto current = first; current->next; current = current->next)
	{
		if (current->next->data == key)
		{
			if (current->next == last)
			{
				last = current;
				last->next = nullptr;
				return;
			}
			current->next = current->next->next;
			current->next->next->prev = current;
			return;
		}
		if (current->next == last)
		{
			std::cout << "Cannot delete the node with key " << key << ".Is not in
                                                                    the list.\n";
			return;
		}
	}
}
template<class T>
void List<T>::reverse()
{
	if (isEmpty())
	{
		std::cout << "Cannot reverse an empty list\n";
		return;
	}
	last = first;
	auto current = first;
	std::shared_ptr<Node<T>> previous{ nullptr }, nextNode{ nullptr };
	while (current != nullptr)
	{
		nextNode = current->next;
		current->next = previous;
		current->prev = nextNode;
		previous = current;
		current = nextNode;
	}
	first = previous;
}
```

Circular doubly linked list

```
template <class T>
```

```
class List
{
public:
	List() = default;
	~List();
	List(const std::initializer_list<T>&);
	bool isEmpty()const;
	void displayForward()const;
	void displayBackward()const;
	void push_back(const T&);
	void insertNodeAfterKey(const T&,const T&);
	void deleteNodeWithKey(const T&);
	void reverse();
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& data) :data{ data }, next{}, prev{} {}
		T data{};
		std::shared_ptr<Node> next{ nullptr };
		std::weak_ptr<Node> prev{ nullptr };
	};
	std::shared_ptr<Node<T>> first{ nullptr };
};
template <class T>
List<T>::~List()
{
	if (isEmpty())
	{
		return;
	}
	auto temp = first;
	do
	{
		first = first->next;
	} while (first != temp);
}
template<class T>
bool List<T>::isEmpty() const
{
	return first == nullptr;
}
template<class T>
void List<T>::displayForward() const
{
	if (isEmpty())
	{
		std::cout << "List is empty.Cannot display forward\n";
		return;
	}
	std::cout << "List is (forward):";
	auto current = first;
	do
	{
		std::cout << current->data << ' ';
		current = current->next;
	} while (current != first);
	std::cout << '\n';
}
template<class T>
void List<T>::displayBackward() const
```

```
{
	if (isEmpty())
	{
		std::cout << "List is empty.Cannot display backward\n";
		return;
	}
	std::cout << "List is (backward):";
	auto current = first->prev.lock();
	do
	{
		std::cout << current->data << ' ';
		current = current->prev.lock();
	} while (current != first->prev.lock());
	std::cout << '\n';
}
template<class T>
void List<T>::push_back(const T& newData)
{
	if (isEmpty())
	{
		first = std::make_shared<Node<T>>(newData);
		first->next = first;
		first->prev = first;
		return;
	}
	auto newNode = std::make_shared<Node<T>>(newData);
	first->prev.lock()->next = newNode;
	newNode->prev = first->prev;
	newNode->next = first;
	first->prev = newNode;
}
template<class T>
List<T>::List(const std::initializer_list<T>& il)
{
	for (auto p = il.begin();p != il.end();p++)
	{
		push_back(*p);
	}
}
template<class T>
void List<T>::insertNodeAfterKey(const T& key, const T& newData)
{
	if (isEmpty())
	{
		std::cout << "List is empty.No key found!\n";
		return;
	}
	auto current = first;
	do
	{
		if (current->data == key)
		{
			auto newNode = std::make_shared<Node<T>>(newData);
			if (current == first)
			{
				if (current->next == first)
				{
					first->next = newNode;
					newNode->prev = first;
					first->prev = newNode;
					newNode->next = first;
					return;
```

```
                        }
                        else
                        {
                                newNode->next = first->next;
                                first->next->prev = newNode;
                                first->next = newNode;
                                newNode->prev = first;
                                return;
                        }
                }
                newNode->next = current->next;
                current->next->prev = newNode;
                current->next = newNode;
                newNode->prev = current;
                return;
        }
        if (current->next == first)
        {
                std::cout << "The node with key " << key << " is not in the list!\n";
                return;
        }
        current = current->next;
    } while (current != first);
}
template<class T>
void List<T>::deleteNodeWithKey(const T& key)
{
    if (isEmpty())
    {
        std::cout << "List is empty.Cannot delete the node with key " << key << '\n';
        return;
    }
    if (first->data == key)
    {
        if (first == first->next)
        {
            first = nullptr;
            return;
        }
        auto temp = first->prev;
        first = first->next;
        first->prev = temp;
        temp.lock()->next = first;
        return;
    }
    auto current = first;
    do
    {
        if (current->data == key)
        {
            current->next->prev = current->prev;
            current->prev.lock()->next = current->next;
            current = nullptr;
            return;
        }
        if (current->next == first)
        {
            std::cout << "Cannot delete the node with key " << key << ".Is not in
                                                                    the list.\n";
            return;
        }
        current = current->next;
```

```
	} while (current != first);
}
template<class T>
void List<T>::reverse()
{
	if (isEmpty())
	{
		std::cout << "Cannot reverse an empty list!\n";
		return;
	}
	auto current = first;
	std::shared_ptr<Node<T>> previous{ first->prev }, nextNode{ nullptr };
	do
	{
		nextNode = current->next;
		current->next = previous;
		current->prev = nextNode;
		previous = current;
		current = nextNode;
	} while (current != first);
	first = previous;
}
```

STACK ADT

```
template <class T>
class Stack
{
public:
	Stack() = default;
	~Stack();
	bool isEmpty()const;
	void push(const T&);
	void pop();
	T peek(int)const;
	void display()const;
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& data) :data{ data }, next{ nullptr } {};
		T data{};
		std::shared_ptr<Node> next{ nullptr };
	};
	std::shared_ptr<Node<T>> top{ nullptr };
};
template<class T>
Stack<T>::~Stack()
{
	while (top != nullptr)
	{
		top = top->next;
	}
}
template<class T>
bool Stack<T>::isEmpty() const
{
	return top == nullptr;
}
template<class T>
void Stack<T>::push(const T& newData)
```

```
{
	auto newNode = std::make_shared<Node<T>>(newData);
	if (newNode == nullptr)
	{
		std::cout << "Stack is full\n";//Heap is full
	}
	else
	{
		newNode->next = top;
		top = newNode;
	}
}
template<class T>
void Stack<T>::pop()
{
	if (isEmpty())
	{
		std::cout << "Stack underflow\n";
	}
	else
	{
		top = top->next;
	}
}
template<class T>
T Stack<T>::peek(int position) const
{
	if (isEmpty())
	{
		std::cout << "Stack is empty.Cannot peek\n";
		return T{};
	}
	else
	{
		if (position <= 0)
		{
			std::cout << "Invalid position\n";
			return T{};
		}
		auto p = top;
		for (int i = 0; p != nullptr && i < position - 1; p = p->next, i++);
		if (p != nullptr)
		{
			return p->data;
		}
		else
		{
			return T{};
		}
	}
}
template<class T>
void Stack<T>::display() const
{
	if (isEmpty())
	{
		std::cout << "Stack is empty.Cannot display\n";
	}
	else
	{
		std::cout << "Stack is:";
		auto p = top;
```

```
		while (p != nullptr)
		{
			std::cout << p->data << ' ';
			p = p->next;
		}
		std::cout << '\n';
	}
}
```

QUEUE ADT

```
template <class T>
class Queue
{
public:
	Queue() = default;
	~Queue();
	Queue(const std::initializer_list<T>& il);
	bool isEmpty()const;
	void enqueue(const T& newData);
	void dequeue();
	void display()const;
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& data) :data{ data }, next{ nullptr }{}
		T data{};
		std::shared_ptr<Node> next{ nullptr };
	};
	std::shared_ptr<Node<T>> front{ nullptr };
	std::shared_ptr<Node<T>> rear{ nullptr };
};

template<class T>
inline Queue<T>::~Queue()
{
	while (front != nullptr)
	{
		front = front->next;
	}
}

template<class T>
inline Queue<T>::Queue(const std::initializer_list<T>& il)
{
	for (auto p = begin(il); p != end(il); ++p)
	{
		enqueue(*p);
	}
}

template<class T>
inline bool Queue<T>::isEmpty() const
{
	return front == nullptr && rear == nullptr;
}

template<class T>
inline void Queue<T>::enqueue(const T& newData)
{
```

```
	if (isEmpty())
	{
		front = std::make_shared<Node<T>>(newData);
		rear = front;
		return;
	}
	auto newNode = std::make_shared<Node<T>>(newData);
	rear->next = newNode;
	rear = newNode;
}

template<class T>
inline void Queue<T>::dequeue()
{
	if (isEmpty())
	{
		std::cout << "Cannot dequeue an empty queue\n";
		return;
	}
	if (front == rear)
	{
		front = rear = nullptr;
		return;
	}
	front = front->next;
}

template<class T>
inline void Queue<T>::display() const
{
	if (isEmpty())
	{
		std::cout << "Cannot display an empty queue\n";
		return;
	}
	auto current = front;
	std::cout << "Queue is: ";
	while (current != nullptr)
	{
		std::cout << current->data << ' ';
		current = current->next;
	}
	std::cout << '\n';
}
```

ARBORI BINARI

//Tree.h

```
template <class T>
class Tree
{
	template <class T>
	struct Node;
public:
	void createTree();
	void printPreOrder() const;
	void printInOrder() const;
	void printPostOrder() const;
	void printLevelOrder() const;
	auto searchNode(const T& key)->std::shared_ptr<Node<T>> const;
	void deleteNode(const T& key);
```

```
	int countNodes() const;//any degree
	int countLeaves() const;//deg(0)
	int height()const;
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& key) : key{ key } {}
		T key;
		std::shared_ptr<Node> left{ nullptr };
		std::shared_ptr<Node> right{ nullptr };

	};
	void printPreOrder(const std::shared_ptr<Node<T>>& node) const;
	void printInOrder(const std::shared_ptr<Node<T>>& node) const;
	void printPostOrder(const std::shared_ptr<Node<T>>& node) const;
	void printLevelOrder(const std::shared_ptr<Node<T>>& node)const;
	auto searchNode(std::shared_ptr<Node<T>>& node, const T& key)
                                                    ->std::shared_ptr<Node<T>> const;
	void setExtremeRightToNull(std::shared_ptr<Node<T>>& node, std::shared_ptr<Node<T>>&
                                                    extremeRight);//... for deleteNode()
	void deleteNode(std::shared_ptr<Node<T>>& node, const T& key);//... for deleteNode()
	int countNodes(const std::shared_ptr<Node<T>>& node) const;
	int countLeaves(const std::shared_ptr<Node<T>>& node) const;
	int height(const std::shared_ptr<Node<T>>& node)const;
	std::shared_ptr<Node<T>> root{ nullptr };
};
template <class T>
void Tree<T>::createTree()
{
	std::shared_ptr<Node<T>> p;
	T data;
	std::queue<std::shared_ptr<Node<T>>> q;
	std::cout << "Enter root value:";
	std::cin >> data;
	root = std::make_shared<Node<T>>(data);
	root->left = nullptr;
	root->right = nullptr;
	q.push(root);
	while (!q.empty())
	{
		p = q.front();
		q.pop();
		std::cout << "Enter left child of " << p->key << ":";
		std::cin >> data;
		if (data != T())
		{
			p->left = std::make_shared<Node<T>>(data);
			p->left->left = nullptr;
			p->left->right = nullptr;
			q.push(p->left);
		}
		std::cout << "Enter right child of " << p->key << ":";
		std::cin >> data;
		if (data != T())
		{
			p->right = std::make_shared<Node<T>>(data);
			p->right->left = nullptr;
			p->right->right = nullptr;
			q.push(p->right);
		}
```

```
	}
}
//Order of traversals are relative to the root
//Preorder root - L - R
template<class T>
void Tree<T>::printPreOrder(const std::shared_ptr<Node<T>>& node) const
{
	if (node != nullptr)
	{
		std::cout << node->key << ", ";
		printPreOrder(node->left);
		printPreOrder(node->right);
	}
}
template<class T>
void Tree<T>::printPreOrder() const
{
	if (root == nullptr)
	{
		std::cout << "Empty BT\n";
	}
	else
	{
		printPreOrder(root);
	}
}
//Inorder L - root - R
template <class T>
void Tree<T>::printInOrder(const std::shared_ptr<Node<T>>& node) const
{
	if (node != nullptr)
	{
		printInOrder(node->left);
		std::cout << node->key << ", ";
		printInOrder(node->right);
	}
}
template <class T>
void Tree<T>::printInOrder() const
{
	if (root == nullptr)
	{
		std::cout << "Empty BT\n";
	}
	else
	{
		printInOrder(root);
	}
}
//Postorder L - R - root
template<class T>
void Tree<T>::printPostOrder(const std::shared_ptr<Node<T>>& node) const
{
	if (node != nullptr)
	{
		printPostOrder(node->left);
		printPostOrder(node->right);
		std::cout << node->key << ", ";
	}
}
template<class T>
void Tree<T>::printPostOrder() const
```

```
{
	if (root == nullptr)
	{
		std::cout << "Empty BT\n";
	}
	else
	{
		printPostOrder(root);
	}
}
template<class T>
void Tree<T>::printLevelOrder(const std::shared_ptr<Node<T>>& node) const
{
	std::queue<std::shared_ptr<Node<T>>> q;
	std::cout << node->key << ", ";
	q.push(node);
	while (!q.empty())
	{
		auto temp = q.front();//(1)take an address
		q.pop();//(1)take it out
		if (temp->left != nullptr) //(2)visit its left child
		{
			std::cout << temp->left->key << ", ";
			q.push(temp->left);
		}
		if (temp->right != nullptr)//(3)visit its right child
		{
			std::cout << temp->right->key << ", ";
			q.push(temp->right);
		}
	}
}
template<class T>
void Tree<T>::printLevelOrder() const
{
	if (root == nullptr)
	{
		std::cout << "Empty BT\n";
	}
	else
	{
		printLevelOrder(root);
	}
}
template<class T>
auto Tree<T>::searchNode(const T& key)->std::shared_ptr<Node<T>> const
{
	return searchNode(root, key);
}
//search of node at the deepest level(if duplicates) and the most right(if duplicates)
template <class T>
auto  Tree<T>::searchNode(std::shared_ptr<Node<T>>& node, const T& key)
                                                ->std::shared_ptr<Node<T>> const
{
	if (node == nullptr)
	{
		return nullptr;
	}
	std::shared_ptr<Node<T>> out = nullptr;
	std::queue<std::shared_ptr<Node<T>>> q;
	q.push(node);
	while (!q.empty())
```

```
	{
		auto temp = q.front();
		q.pop();
		if (temp->key == key)
		{
			out = temp;
			std::cout << "\nHIT\n";
		}
		if (temp->left != nullptr)
		{
			q.push(temp->left);
		}
		if (temp->right != nullptr)
		{
			q.push(temp->right);
		}
	}
	return out;
}
template <class T>
void Tree<T>::setExtremeRightToNull(std::shared_ptr<Node<T>>& node,
                                    std::shared_ptr<Node<T>>& extremeRight)
{
	std::queue<std::shared_ptr<Node<T>>> q;
	q.push(node);
	std::shared_ptr<Node<T>> temp = nullptr;
	while (!q.empty())
	{
		temp = q.front();
		q.pop();
		if (temp->left != nullptr)
		{
			if (temp->left == extremeRight)
			{
				std::cout << "FOUND\n";
				temp->left = nullptr;
				return;
			}
			else
			{
				q.push(temp->left);
			}
		}
		if (temp->right != nullptr)
		{
			if (temp->right == extremeRight)
			{
				std::cout << "FOUND\n";
				temp->right = nullptr;
				return;
			}
			else
			{
				q.push(temp->right);
			}
		}

	}
}
template <class T>
void Tree<T>::deleteNode(std::shared_ptr<Node<T>>& node, const T& key)
{
```

```
	auto nodeToDelete = searchNode(key);
	if (nodeToDelete != nullptr)
	{
		std::queue<std::shared_ptr<Node<T>>> q;
		q.push(node);
		std::shared_ptr<Node<T>> temp = nullptr;
		while (!q.empty())
		{
			temp = q.front();
			q.pop();
			if (temp->left != nullptr)
			{
				q.push(temp->left);
			}
			if (temp->right != nullptr)
			{
				q.push(temp->right);
			}
		}
		T keyAtDeepestRight = temp->key;
		setExtremeRightToNull(node, temp);
		nodeToDelete->key = keyAtDeepestRight;
	}
}
template<class T>
void Tree<T>::deleteNode(const T& key)
{
	deleteNode(root, key);
}
template<class T> //Done in post order, the MOST used when proccessing BT
int Tree<T>::countNodes(const std::shared_ptr<Node<T>>& node) const
{
	int x, y;
	if (node != nullptr)
	{
		x = countNodes(node->left);
		y = countNodes(node->right);
		return x + y + 1;
	}
	return 0;
}
template<class T>
int Tree<T>::countNodes() const
{
	return countNodes(root);
}
template<class T>
int Tree<T>::countLeaves(const std::shared_ptr<Node<T>>& node) const
{
	int x, y;
	if (node != nullptr)
	{
		x = countLeaves(node->left);
		y = countLeaves(node->right);
		if (node->left == nullptr && node->right == nullptr) //(#)
		{
			return x + y + 1;//count it
		}
		else
		{
			return x + y;//don't count it
		}
```

```
	}
	return 0;
}
template<class T>
int Tree<T>::countLeaves() const
{
	return countLeaves(root);
}
//Same counting procedure, done in post order,
//for different degrees => different conditions(#)
//deg(2)            if(node->left && node->right)
//deg(1) or deg(2)  if(node->left || node->right)
//deg(1)            if((node->left && !node->right) || (!node->left && node->right))
template<class T>
int Tree<T>::height(const std::shared_ptr<Node<T>>& node) const
{
	int x = 0, y = 0;
	if (node == nullptr)
	{
		return 0;
	}
	x = height(node->left);
	y = height(node->right);
	if (x > y)
	{
		return x + 1;
	}
	else
	{
		return y + 1;
	}
}
template<class T>
int Tree<T>::height() const
{
	return height(root) - 1;//!!!
}
```

//Binary Tree.cpp

```
int main()
{
    Tree<int> BT;
    BT.createTree();
    //    1
    //   / \
    //  2   3
    // / \  / \
    //4   5 6  7
    std::cout << "PreOrder\n";
    BT.printPreOrder();//1 2 4 5 3 6 7
    std::cout << "\nInOrder\n";
    BT.printInOrder();//4 2 5 1 6 3 7
    std::cout << "\nPostOrder\n";
    BT.printPostOrder();//4 5 2 6 7 3 1
    std::cout << "\nLevelOrder\n";
    BT.printLevelOrder();//1 2 3 4 5 6 7
    auto found = BT.searchNode(5);
    if (found != nullptr)
    {
        std::cout << "Found " << found->key << " in the tree";
    }
    std::cout << "\nNumber of nodes:" << BT.countNodes();//any degree
    std::cout << "\nNumber of leaves:" << BT.countLeaves();//deg(0)
```

```
    std::cout << "\nHeight is:" << BT.height();
    BT.deleteNode(2);
    //    1
    //   / \
    //  7   3
    // / \  /
    //4   5 6
    std::cout << "\nAfter deletion\n";
    BT.printLevelOrder();//1 7 3 4 5 6
}
```

ARBORI BINARI DE CAUTARE

//Tree.h

```
template <class T>
class Tree
{
	template <class T>
	struct Node;
public:
	~Tree();
	void printInOrder() const;
	void insert(const T& val);
	void insert(T&& val);
	auto search(const T& val)->std::shared_ptr<Node<T>> const;
	void remove(const T& val);
private:
	template <class T>
	struct Node
	{
		Node() = default;
		explicit Node(const T& key) : key{ key } {}
		explicit Node(T&& key) : key{ std::move(key) } {}
		T key;
		std::shared_ptr<Node> left{ nullptr };
		std::shared_ptr<Node> right{ nullptr };
	};
	void print(const std::shared_ptr<Node<T>>& node) const;
	void insert(std::shared_ptr<Node<T>>& node, const T& val);
	void insert(std::shared_ptr<Node<T>>& node, T&& val);
	auto search(std::shared_ptr<Node<T>>& node, const T& val)->std::shared_ptr<Node<T>>
                                                                    const;
	void remove(std::shared_ptr<Node<T>>& node, const T& val);
	std::shared_ptr<Node<T>> root{ nullptr };
};
template<class T>
Tree<T>::~Tree()
{
	while (root != nullptr)
	{
		remove(root->key);
	}
}
template <class T>
void Tree<T>::print(const std::shared_ptr<Node<T>>& node) const
{
	if (node != nullptr)
	{
		print(node->left);
		std::cout << node->key << ", ";
		print(node->right);
	}
}
```

```
template <class T>
void Tree<T>::printInOrder() const
{
	if (root == nullptr)
	{
		std::cout << "Empty BST\n";
	}
	else
	{
		print(root);
	}
}
template <class T>
void Tree<T>::insert(std::shared_ptr<Node<T>>& node, const T& val) {

	if (node == nullptr)
	{
		node = std::make_shared<Node<T>>(val);
	}
	else
	{
		if (val < node->key)
		{
			insert(node->left, val);
		}
		else if (val > node->key)
		{
			insert(node->right, val);
		}
		else
		{
			std::cout << "Warning: Value " << node->key << " already exists, so
                                                        nothing will be done.\n";
		}
	}
}
template <class T>
void Tree<T>::insert(const T& val)
{
	insert(root, val);
}
template <class T>
void Tree<T>::insert(std::shared_ptr<Node<T>>& node, T&& val) {

	if (node == nullptr)
	{
		node = std::make_shared<Node<T>>(std::move(val));
	}
	else
	{
		if (val < node->key)
		{
			insert(node->left, val);
		}
		else if (val > node->key)
		{
			insert(node->right, val);
		}
		else
		{
			std::cout << "Warning: Value " << node->key << " already exists, so
                                                        nothing will be done.\n";
```

```
		}
	}
}
template <class T>
void Tree<T>::insert(T&& val)
{
	insert(root, std::move(val));
}
template <class T>
auto Tree<T>::search(std::shared_ptr<Node<T>>& node, const T& val)->std::shared_ptr<Node<T>>
                                                                                const
{
	if (node == nullptr || node->key == val)
	{
		return node;
	}
	else if (val < node->key)
	{
		return search(node->left, val);
	}
	return search(node->right, val);
}
template <class T>
auto Tree<T>::search(const T& val)->std::shared_ptr<Node<T>> const
{
	return search(root, val);
}
template<class T>
void Tree<T>::remove(std::shared_ptr<Node<T>>& node, const T& val)
{
	if(node && val < node->key)
		remove(node->left,val);
	else if(node && val > node->key)
		remove(node->right,val);
	else if(node && node->key == val)
	{
		if(!node->left)
			node = node->right;
		else if(!node->right)
			node = node->left;
		else
		{
			auto temp = node->left;
			while(temp->right)
				temp = temp->right;//In Order Predecessor
			node->key = temp->key;
			remove(node->left, temp->key);
			//apelata recursiv pt. oricate noduri tb. sterse
			//adica, cazul cand tb. facute m.m. modificari in BST
		}
	}
	else
	{
		std::cout << "The value " << val << " is not in the tree\n";
	}

}
template<class T>
void Tree<T>::remove(const T& val)
{
	remove(root,val);
}
```

//Binary Search Tree.cpp

```
int main()
{
    Tree<int> BST;
    BST.insert(10);
    BST.insert(2);
    BST.insert(50);
    BST.insert(51);
    BST.insert(42);
    int x = 1;
    BST.insert(x);
    BST.insert(3);
    //    10
    //   /  \
    //  2    50
    // / \   / \
    //1  3  42 51
    BST.printInOrder();
    BST.insert(11);
    //    10
    //   /  \
    //  2    50
    // / \   / \
    //1  3  42 51
    //      /
    //     11
    std::cout << '\n';
    BST.printInOrder();
    auto found = BST.search(50);
    if (found)
    {
        std::cout << "\nElement " << found->key << " found in the tree\n";
    }
    std::cout << '\n';
    BST.remove(10);
    BST.printInOrder();
}
```

ARBORI AVL

```
template <class T>
class Tree
{
	template <class T>
	struct Node;
public:
	~Tree();
	void printInOrder() const;
	void insert(const T& val);
	void remove(const T& val);
private:
	template <class T>
	struct Node
	{
		T key;
		int height{ 0 };
		std::shared_ptr<Node> left{ nullptr };
		std::shared_ptr<Node> right{ nullptr };
		explicit Node(const T& key) : key{ key } {}
	};
	void print(const std::shared_ptr<Node<T>>& node) const;
	int nodeHeight(std::shared_ptr<Node<T>>& node) const;//update height
```

```
	int balanceFactor(std::shared_ptr<Node<T>>& node) const;
	void LLRotation(std::shared_ptr<Node<T>>& p);
	void LRRotation(std::shared_ptr<Node<T>>& p);
	void RLRotation(std::shared_ptr<Node<T>>& p);
	void RRRotation(std::shared_ptr<Node<T>>& p);
	void insert(std::shared_ptr<Node<T>>& node, const T& val);
	void remove(std::shared_ptr<Node<T>>& node, const T& val);
	std::shared_ptr<Node<T>> root{ nullptr };
};
template<class T>
Tree<T>::~Tree()
{
	while (root != nullptr)
	{
		remove(root->key);
	}
}
template <class T>
void Tree<T>::print(const std::shared_ptr<Node<T>>& node) const
{
	if (node != nullptr)
	{
		print(node->left);
		std::cout << node->key << ", ";
		print(node->right);
	}
}
template <class T>
void Tree<T>::printInOrder() const
{
	if (root == nullptr)
	{
		std::cout << "Empty AVL Tree\n";
	}
	else
	{
		print(root);
	}
}
template <class T>
int Tree<T>::nodeHeight(std::shared_ptr<Node<T>>& node) const
{
	int hl = node && node->left ? node->left->height : 0;
	int hr = node && node->right ? node->right->height : 0;
	return hl > hr ? hl + 1 : hr + 1;
}
template <class T>
int Tree<T>::balanceFactor(std::shared_ptr<Node<T>>& node) const
{
	int hl = node && node->left ? node->left->height : 0;
	int hr = node && node->right ? node->right->height : 0;
	return hl - hr;
}
template <class T>
void Tree<T>::LLRotation(std::shared_ptr<Node<T>>& p)
{
	auto pl = p->left;
    auto plr = pl->right;

	pl->right = p;
	p->left = plr;
```

```
	p->height = nodeHeight(p);//update height
	pl->height = nodeHeight(pl);//update height

	if (root == p)
	{
		root = pl;
	}
	else
	{
		p = pl;
	}
}
template <class T>
void Tree<T>::LRRotation(std::shared_ptr<Node<T>>& p)
{
	auto pl = p->left;
	auto plr = pl->right;

	pl->right = plr->left;
	p->left = plr->right;

	plr->left = pl;
	plr->right = p;

	pl->height = nodeHeight(pl);
	p->height = nodeHeight(p);
	plr->height = nodeHeight(plr);

	if (root == p)
	{
		root = plr;
	}
	else
	{
		p = plr;
	}
}
template <class T>
void Tree<T>::RLRotation(std::shared_ptr<Node<T>>& p)
{
	auto pr = p->right;
	auto prl = pr->left;

	p->right = prl->left;
	pr->left = prl->right;

	prl->left = p;
	prl->right = pr;

	p->height = nodeHeight(p);
	pr->height = nodeHeight(pr);
	prl->height = nodeHeight(prl);

	if (root == p)
	{
		root = prl;
	}
	else
	{
		p = prl;
	}
}
```

```
template <class T>
void Tree<T>::RRRotation(std::shared_ptr<Node<T>>& p)
{
	auto pr = p->right;
	auto prl = pr->left;

	pr->left = p;
	p->right = prl;

	p->height = nodeHeight(p);
	pr->height = nodeHeight(pr);

	if (root == p)
	{
		root = pr;
	}
	else
	{
		p = pr;
	}
}
template <class T>
void Tree<T>::insert(std::shared_ptr<Node<T>>& node, const T& val) {

	if (node == nullptr)
	{
		node = std::make_shared<Node<T>>(val);
	}
	else
	{
		if (val < node->key)
		{
			insert(node->left, val);
		}
		else if (val > node->key)
		{
			insert(node->right, val);
		}
		else
		{
			std::cout << "Warning: Value " << node->key << " already exists, so
                                                    nothing will be done.\n";
		}
	}
	node->height = nodeHeight(node);
	if (balanceFactor(node) == 2 && balanceFactor(node->left) == 1)
	{
		LLRotation(node);
	}
	else if (balanceFactor(node) == 2 && balanceFactor(node->left) == -1)
	{
		LRRotation(node);
	}
	else if (balanceFactor(node) == -2 && balanceFactor(node->right) == -1)
	{
		RRRotation(node);
	}
	else if (balanceFactor(node) == -2 && balanceFactor(node->right) == 1)
	{
		RLRotation(node);
	}
}
```

```
template <class T>
void Tree<T>::insert(const T& val)
{
	insert(root, val);
}
template <class T>
void Tree<T>::remove(std::shared_ptr<Node<T>>& node, const T& val)
{
	if (node && val < node->key)
		remove(node->left, val);
	else if (node && val > node->key)
		remove(node->right, val);
	else if (node && node->key == val)
	{
		if (!node->left)
			node = node->right;
		else if (!node->right)
			node = node->left;
		else
		{
			auto temp = node->left;
			while (temp->right)
				temp = temp->right;//In Order Predecessor
			node->key = temp->key;
			remove(node->left, temp->key);
			//apelata recursiv pt. oricate noduri tb. sterse
			//adica, cazul cand tb. facute m.m. modificari in AVLT
		}
	}
	else
	{
		std::cout << "The value " << val << " is not in the tree\n";
	}
	//nodul precedent este cel care a devenit imbalanced
	//deci la returning time, node este nodul precedent
	if (balanceFactor(node) == 2 && balanceFactor(node->left) == 1)
	{
		LLRotation(node); //L1
	}
	else if (balanceFactor(node) == 2 && balanceFactor(node->left) == -1)
	{
		LRRotation(node); //L-1
	}
	else if (balanceFactor(node) == 2 && balanceFactor(node->left) == 0)
	{
		LLRotation(node); //L1 sau L-1, am ales L1
	}
	else if (balanceFactor(node) == -2 && balanceFactor(node->right) == -1)
	{
		RRRotation(node); //R-1
	}
	else if (balanceFactor(node) == -2 && balanceFactor(node->right) == 1)
	{
		RLRotation(node); //R1
	}
	else if (balanceFactor(node) == -2 && balanceFactor(node->right) == 0)
	{
		RRRotation(node); //R1 sau R-1, am ales R-1
	}
}
template<class T>
void Tree<T>::remove(const T& val)
```

```
{
	remove(root, val);
}
```