16. Структури от данни. Стек, опашка, списък, дърво. Основни операции върху тях. Реализация.

Структури от данни - дефиниране. Линейни структури от данни – списък, опашка, стек. Логическо описание. Статични и динамични реализации. Дефиниране на класове, реализиращи статично или динамично свързан списък (едностранно или двустранно), опашка или стек. Дървовидни структури от данни – двоично кореново дърво и двоично кореново дърво за бързо търсене. Логическо описание. Статични и динамични реализации. Дефиниране на класове, реализиращи двоично кореново дърво или двоично кореново дърво за бързо търсене.
Забележка. За изпита ще бъдат избрани една от линейните и една от дървовидните структури

Под **структура от данни** се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на програма. За да се определи една структура от данни е необходимо да се направи:

- логическо описание на структурата, което я описва на базата на декомпозицията и на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции;
- физическо представяне на структурата, което дава метода за представяне на структурата в паметта на компютъра.

Стекът е линейна динамична структура от данни. Стекът е крайна редица от елементи от един и същи тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича **връх** на стека. Възможен е пряк достъп само до елемента, който се намира във върха на стека. При тази организация на логическите операции, последният включен елемент се изключва пръв. Затова стекът се определя още като структура "последен влязъл – пръв излязъл" (last in – first out, LIFO).
Широко се използват два основни начина за представяне на стек: **последователно** и **свързвано**. При последователното представяне, предварително в паметта се запазва блок, вътре в който се помества стекът и той там расте и се съкращава. При включване на елементи в стека, те се поместват в последователни адреси в неизползваната част веднага след върха на стека.
При свързаното представяне последователните елементи се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в стека, стойността на този указател трябва да бъде някаква различима стойност (например NULL). За задаване на стека е достатъчен указател към върха на стека. В общия случай елементите на стека при свързаното представяне се състоят от две полета – inf (данните, записани в елемента) и link (указател към следващия елемент). Сега ще дефинираме клас, който реализира свързаното представяне на стек.
Първо ще дефинираме помощен клас Item, който реализира двойната кутия, чрез която се представят елементите на стека. За по-голяма общност, дефинираме класовете като шаблони.

```
template <class T> class Stack;
template <class T> class Item {
        friend class Stack <T>;
        private:
                Item (const T& x = 0) {
                        inf = x;
                        link = NULL;
                }
                T inf;
                Item *link;
};
```

Тъй като класът Item използва класът Stack в декларацията си, затова прототипът на класа Stack предшества декларацията на класа Item. Всички членове на Item са капсулирани. Чрез декларацията за приятелство, класът Item позволява само на класа Stack да създава и обработва негови обекти. Сега вече сме готови да дефинираме класът Stack. Тъй като стекът се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
class Stack {
public:
	Stack (const T&);
	Stack ();
	~Stack ();
	Stack (const Stack &);
	Stack& operator= (const Stack &);
	void push (const T&);
	bool pop (T&);
	bool top(T&) const;
	bool empty () const;
private:
	Item<T> *start;
	void delStack();
	void copyStack (const Stack &);
};
template <class T> Stack<T>::Stack (const T& x){
	start = new Item<T> (x);
}
template <class T> Stack<T>::Stack () {
	start = NULL;
}
template <class T> Stack<T>::~Stack (){
	delStack();
}
template <class T> Stack<T>::Stack (const Stack<T> &r){
	copyStack(r);
}
template <class T> Stack<T>& operator= (const Stack<T> &r){
	if (this != &r) {
		delStack ();
		copyStack (r);
	}
	return *this;
}
template <class T> void Stack<T>::delStack () {
	Item<T> *p;
	while (start){
		p = start;
		start = start -> link;
		delete p;
	}
}

template <class T> void Stack<T>::copyStack (const Stack<T> &r){
	if (!r.start) start = NULL;
	else {
		Item<T> *p = r.start, *q, *s;
		start = new Item<T>(p -> inf);
		q = start;
		p = p -> link;
		while (p) {
			s = new Item<T> (p -> inf);
			q -> link = s;
			q = s;
			p = p -> link;
		}
	}
}
```

```
template <class T> void Stack<T>::push (const T& x){
	Item<T> *p = new Item<T>(x);
	p -> link = start;
	start = p;
}
template <class T> bool Stack<T>::pop (T& x){
	if (!start) return false;
	Item<T> *p = start;
	x = start -> inf;
	start = start -> link;
	delete p;
	return true;
}
template <class T> bool Stack<T>::top (T& x) const{
	if (!start) return false;
	x = start -> inf;
	return true;
}
template <class T> bool Stack<T>::empty () const {
	return start == NULL;
}
```

Опашката е крайна редица от елементи от един и същи тип. Операцията включване е допустима за елементите от единия край на редицата, който се нарича **край** на опашката, а операцията изключване на елемент – само за елементите от другия край на редицата, който се нарича **начало** на опашката. Възможен е пряк достъп само до елемента, намиращ се в началото на опашката. При тази организация на логическите операции, пръв се изключва най-отдавна включеният елемент. Затова опашката се определя още като структура от данни "пръв влязъл – пръв излязъл" (first in – first out, FIFO).
Широко се използват два основни начина за физическо представяне на опашка: **последователно** и **свързано**.
При последователното представяне първоначално в паметта се запазва блок, вътре в който опашката да расте и да се съкращава. Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част веднага след края на опашката. Обикновено се счита, че блокът от памет е цикличен – когато краят на опашката достигне края на разпределения блок памет, но има освободена памет в неговото начало, там може да се извърши включване на елементи.
При свързаното представяне последователните елементи на опашката се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в опашката, стойността на този указател трябва да бъде някаква различима стойност (например NULL). За задаване на опашката са достатъчни указатели към началото и края на опашката. В общия случай елементите на опашката при свързаното представяне се състоят от две полета – inf (данните, записани в елемента) и link (указател към следващия елемент). Сега ще дефинираме клас, който реализира свързаното представяне на опашка. Първо ще дефинираме помощен клас Item, който реализира двойната кутия, чрез която се представят елементите на опашката.
За по-голяма общност, дефинираме класовете като шаблони.

```
template <class T> class Queue;
template <class T> class Item {
	friend class Queue <T>;
	private:
		Item (const T& x = 0){
			inf = x;
			link = NULL;
		}
		T inf;
		Item *link;
};
```

Тъй като класът Item използва класът Queue в декларацията си, затова прототипът на класа Queue предшества декларацията на класа Item. Всички членове на Item са капсулирани. Чрез декларацията за приятелство, класът Item позволява само на класа Queue да създава и обработва негови обекти. Сега вече сме готови да дефинираме класът Queue. Тъй като опашката се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T> class Queue {
public:
	Queue(const T&);
	Queue();
	~Queue();
	Queue (const Queue &);
	Queue& operator= (const Queue &) ;
	void InsertElem (const T&);
	bool DeleteElem (T &);
	bool ViewElem(T &) const;
	bool empty() const;
private:
	Item<T> *front, *rear;
	void delQueue();
	void copyQueue (const Queue &);
} ;
template <class T> Queue<T>::Queue(const T& x){
	front = new Item<T>(x);
	rear = front;
}
template <class T> Queue<T>::Queue() {
	front = rear = NULL;
}
template <class T> Queue<T>::~Queue() {
	 delQueue();
}
template <class T> Queue<T>::Queue (const Queue<T> &r){
	copyQueue (r);
}
template <class T> Queue<T>& Queue<T>::operator= (const Queue<T> &r){
	if (this != &r){
		delQueue();
		copyQueue(r);
	}
	return *this;
}
template <class T> void Queue<T>::delQueue() {
	T x;
	while (DeleteElem(x));
}
template <class T> void Queue<T>::copyQueue(const Queue<T> &r) {
	front = rear = NULL;
	 if (r.front) {
		Item <T> *p = r.front;
		while (p) {
			InsertElem (p -> inf);
			p = p -> link;
		}
	}
}
template <class T> void Queue<T>::InsertElem (const T& x){
	Item <T> *p = new Item<T>(x);
	if (front) { rear->link = p; rear = p; }
	else front = rear = p;
}
template <class T> bool Queue<T>::DeleteElem (T &x){
	if (!front) return false;
	Item <T> *p = front;
	x = p -> inf;
	if (front == rear) front = rear = NULL;
	else front = front -> link;
	delete p;
	return true;}
```

```
template <class T> bool Queue<T>::ViewElem(T &x) const {
        if (!front) return false;
        x = front -> inf;
        return true;
}
template <class T> bool Queue<T>::empty() const{
return front == NULL;
}
```

Последователна (статична) реализация на стек и опашка

Най-простият начин за имплементацията на списък е елементите му да се съхраняват *последователно* в паметта на компютъра. В този случай адресът на *k*-тия елемент *addr*(*k*) се определя по формулата *addr*(*k*) = *addr*(*k*–1) + *sizeof*(*data*), където *sizeof*(*data*) е паметта, необходима за съхранени на един елемент от списъка. При този вид имплементация достъпът до *k*-тия елемент от списъка е *пряк* (затова тази имплементация освен *последователна*, често се нарича и *статична*).

Удобството от директния достъп до елементите се компенсира с неефективно включване и изключване на елемент: за да включим елемент след *k*-тия, или да изключим *k*-тия елемент, трябва да изместим всички елементи от *k*+1 до *n* с една позиция наляво. Търсенето по ключ също е бавна операция: в средния случай сложността [й] (както и сложността на вмъкването и изтриването) е *O*(*n*/2), т.е. O(*n*).

Ще реализираме структурите стек и опашка чрез последователно заемане на памет.

В случая със стек ще въведем единствен указател, който ще сочи адреса на върха на стека.Ще дефинираме макрос MAX, който задава максималния брой елементи в стека. При включване на нов елемент се проверява дали top не е станало по-голямо от MAX и ако е така, значи има препълване. В програмата по-долу е направена още една промяна — преди всеки опит за изключване на елемент се прави проверка дали стекът не е празен.

```
#include <stdio.h>
#define MAX 10
typedef int data;
data stack[MAX];
int top;
void init() { top=0; }
void push(data i) {
        if (top==MAX) printf("Препълване на стека! \n");
        else stack[top++]=i;
}
data pop() {
        if (top==0) { printf("Стекът е празен! \n"); return -1; }
        else return stack[--top];
}
int empty() { return (top==0); }
void main() {
        data p;
        init();
        // Прочитат се цели числа от клавиатурата до прочитане на 0 и се включват в стека
        do {
                scanf("%d",&p);
                if (p!=0) push(p);
        } while (p!=0);
          /* Изключват се последователно всички елементи от стека и се печатат. Това
             ще доведе до отпечатване на първоначално въведената последователност в
             обратен ред */
        while (!empty()) printf("%d ",pop());
        printf("\n");
}
```

Имплементацията на опашка е малко по-сложна. За последователното резервиране на памет ще използваме отново масив, но указателите са два: front, който сочи началото на опашката и rear, сочещ позицията след края на опашката.

Включването и изключването на елементи се осъществява по следния начин:

- *Включване на елемент i*: Записваме *i* на позицията, сочена от rear, и увеличаваме указателя rear.
- *Изключване на елемент*: Връщаме елемента на позиция front и увеличаваме указателя front.

При реализацията на включването и изключването трябва да се справим с няколко проблема. Ако многократно включваме и изключваме елементи в опашката, то указателите front и rear ще се увеличават непрекъснато и бързо ще надхвърлят първоначално заделената за опашката памет. Така, ако използваме масив queue[MAX] (който ще представя

опашка с максимален брой елементи MAX) размерът му бързо ще се окаже недостатъчен, въпреки, че на практика в опашката ще има по-малко от MAX елемента. Затова, ако някой от двата указателя front или rear стане равен на MAX, то ще му се присвоява стойност 0. Така се постигаме "цикличност" на масива queue.
В началото указателите front и rear сочат нулевият елемент. По нататък, ако в даден момент те отново станат равни (сочат една и съща клетка) това означава едно от двете:

- Ако равенството на двата указателя се е получило след изключване на елемент (указателят front е застигнал указателя rear), то опашката остава празна.
- Ако равенството на двата указателя се е получило след включване на елемент (указателят rear е застигнал указателя front), то масивът е пълен (в опашката вече има MAX елемента и повече не могат да се добавят).

За статуса на опашката (дали тя е пълна или празна) ще използваме допълнителна променлива empty, равна на 1, когато опашката е празна и на 0 — в противен случай.

```
#include <stdio.h>
#define MAX 10
typedef int data;
data queue[MAX];
int front, rear, empty;
void init() {
        front=0; rear=0; empty=1;
}
void put(data i) {
        if (front==rear !=empty) { // Проверка за препълване
                /* препълване - указателите са равни, а опашката не е празна */
                printf(" overflow \n");
                return;
        }
        queue[rear++]=i;
        if (rear >= MAX) rear=0;
        empty=0;
}
data get() {
        if (empty) { // Проверка за празна опашка
                printf(" empty! \n");
                return -1;
          }
        data x = queue[front++];
        if (front>= MAX) front=0;
        if (front == rear) empty=1;
        return x;
}
void main() {
        data p;
        init();
        for (int i=0; i<2*MAX; i++) {
                put(i);
                get();
                printf("%d ",i);
        }
        printf("\n");
        // Това ще причини препълване при последното включване
        for (i=0; i<MAX+1; i++) put(i);
        // Това ще причини грешка за празна опашка при последното изключване
         for (i=0; i<MAX+1; i++) get();
}
```

Списъкът е крайна редица от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими на произволно място в редицата. Възможен е достъп до всеки елемент на редицата (пряк или непряк в зависимост от реализацията на списъка).
Има два основни начина за представяне на списък в паметта на компютъра: **свързано** представяне **с една връзка**, **свързано** представяне **с две връзки** и **последователно** представяне.
При последователното представяне списъкът се представя чрез масив, т.е. елементите на списъка са последователно разположени в паметта, но това представяне не се използва, тъй като включването и изключването на елемент е неефективно и освен това заради добре развитите средства за динамично разпределение на паметта.

При свързаното представяне с една връзка последователните елементи на списъка се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в списъка, стойността на този указател трябва да бъде някаква различима стойност (например NULL). За задаване на списъка е достатъчен указател към неговото начало. За удобство при реализирането на операциите включване, изключване и обхождане се въвеждат още указатели към края и към текущ елемент на списъка. В общия случай елементите на списъка при свързаното представяне с една връзка се състоят от две полета – inf (данните, записани в елемента) и link (указател към следващия елемент).
При свързаното представяне с две връзки последователните елементи на списъка се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзките между отделните елементи се осъществяват чрез два указателя към следващия и към предхождащия елемент. Ако елементът е последен в списъка, стойността на указателя към следващия елемент трябва да бъде някаква различима стойност (например NULL), аналогично за указателя към предхождащия елемент за елемента в началото на списъка. За задаване на списъка е достатъчен указател към неговото начало. За удобство при реализирането на операциите включване, изключване и обхождане се въвеждат още указатели към края и указател към текущ елемент на списъка. В общия случай елементите на списъка при свързаното представяне с две връзки се състоят от три полета – inf (данните, записани в елемента), next (указател към следващия елемент) и prev (указател към предхождащия елемент).

Сега ще дефинираме клас, който реализира свързаното представяне на списък с една връзка.
Първо ще дефинираме помощна структура Item, която реализира двойната кутия, чрез която се представят елементите на списъка.За по-голяма общност, дефинираме структурата Item и класа LList като шаблони.

```
template <class T> struct Item {
        T inf;
        Item *link;
};
```

Сега вече сме готови да дефинираме класът LList. Тъй като едносвързаният списък се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T> class LList {
 public:
        LList(const T&);
        LList();
        ~LList();
        LList(const LList &);
        LList& operator= (const LList &);
         void IterStart (Item <T>* = NULL);
        Item <T>* Iter();
        bool IterView(T &) const;
         void InsertToEnd (const T&);
        void InsertToBegin (const T&);
        void InsertAfter (Item <T>*, const T&);
        void InsertBefore (Item <T>*, const T&);
        void DeleteAfter (Item <T>*, T &);
        void DeleteBefore (Item <T>*, T &);
        void DeleteElement (Item <T>*, T &);
        int length() const;
        void print() const;
        void concat (const LList &);
        void reverse();
        bool empty() const;
private:
         Item<T> *start, *end, *current;
        void delLList();
        void copyLList(const LList &);
};
template <class T> LList<T>::LList (const T& x){
        start = new Item<T>;
        start -> inf = x;
        start -> link = NULL;
        end = start;
}
```

```
	template <class T> Llist<T>::LList() {
		start = end = NULL; }
	template <class T> Llist<T>::~LList(){
		delLList();
	}
	template <class T> LList<T>::LList (const LList<T> &r){
		copyLList (r);
	}
	template <class T> LList<T>& LList<T>::operator= (const LList<T> &r){
		if (this != &r){
			delLList();
			copyLList(r);
		}
		return *this;
	}
	template <class T> void LList<T>::delLList(){
		Item <T> *p;
		while (start) {
			p = start;
			start = start -> link;
			delete p;
		}
		end = NULL;
	}
	template <class T> void LList<T>::copyLList(const LList<T> &r){
		start = end = NULL;
		Item <T> *p = r.start;
		while (p){
			InsertToEnd (p -> inf);
			p = p -> link;
		}
	}
	template <class T> void LList<T>::IterStart (Item <T> *p){
		if (p) current = p;
		else current = start;
	}
	template <class T> Item <T>* Llist<T>::Iter(){
		Item <T> *p = current;
		if (current) current = current -> link;
		return p;
	}
	template <class T> bool LList<T>::IterView(T &x){
		if (!current) return false;
		x = current -> inf;
		return true;
}
	template <class T> void LList<T>::InsertToEnd(const T& x){
		Item <T> *p = new Item<T>;
		p -> inf = x;
		p -> link = NULL;
		if (!start) start = end = p;
		else { end -> link = p; end = p; }
	}
	template <class T> void LList<T>::InsertToBegin(const T& x){
		Item <T> *p = new Item<T>;
		p -> inf = x;
		p -> link = start;
		start = p;
		if (!end) end = p;
	}
```

```
template <class T> void LList<T>::InsertAfter (Item <T> *p, const T& x){
	Item <T> *q = new Item<T>;
	q -> inf = x;
	q -> link = p -> link;
	p -> link = q;
	if (p == end) end = q;
}
template <class T> void LList<T>::InsertBefore (Item <T> *p, const T& x){
	Item <T> *q = new Item<T>;
	*q = *p;
	p -> inf = x;
	p -> link = q;
	if (end == p) end = q;
}
template <class T> void LList<T>::DeleteBefore (Item <T> *p, T &x){
	Item <T> *q = start;
	if (q -> link == p) {
		x = q -> inf;
		start = start -> link;
		delete q;
	}
	else {
		while (q -> link -> link != p) q = q -> link;
		DeleteAfter (q, x);
	}
}
template <class T> void LList<T>::DeleteAfter (Item <T> *p, T &x){
	Item <T> *q = p -> link;
	x = q -> inf;
	p -> link = q -> link;
	if (end == q) end = p;
	delete q;
}
template <class T> void LList<T>::DeleteElement (Item <T> *p, T &x){
	if (p == start) {
		x = p -> inf;
		if (start == end) start = end = NULL;
		else start = start -> link;
		delete p;
	}
	else if (p == end) {
		Item <T> *q = start;
		while (q -> link != end) q = q -> link;
		q -> link = NULL;
		delete end;
		end = q;
	}
	else DeleteBefore(p -> link, x); }
template <class T> void LList<T>::print () const{
	Item <T> *p = start;
	while (p){
		cout << p -> inf << ' ';
		p = p -> link;
	}
	cout << endl;
}
template <class T> int LList<T>::length () const{
	Item <T> *p = start;
	int n = 0;
	while (p){
		 n++; p = p -> link;
	}
	return n; }
```

```
template <class T> void LList<T>::concat (const LList<T> &r){
	Item <T> *p = r.start;
	while (p) {
		InsertToEnd (p -> inf);
		p = p -> link;
	}
}
template <class T> void LList<T>::reverse (){
	Item <T> *p, *q, *temp;
	if (start == end) return;
	p = start;
	q = NULL;
	start = end;
	end = p;
	while (p) {
		temp = p -> link;
		p -> link = q;
		q = p;
		p = temp;
	}
}
template <class T> bool LList<T>::empty () const{
	return start == NULL;
}
```

Функциите Iter, IterStart и IterView реализират т.н. **итератор**. Итераторът е указател към текущ елемент на редица. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:
++ - приложена към итератор, премества този итератор да сочи към следващия елемент на редицата;
-- - приложена към итератор, премества този итератор да сочи към предишния елемент на редицата;
* - приложена към итератор, дава елемента, към който сочи итератора.
В шаблона LList, итераторът е current. Функцията IterStart инициализира итератора да сочи към началото на списъка, ако за нея не е указан аргумент или е указан аргумент NULL. Ако за нея е указан друг аргумент, той е началната стойност на итератора. Операцията ++ се реализира от функцията Iter, която между другото връща стария итератор. Операцията -- не е реализирана, тъй като списъкът е само с една връзка и тя би била неефективна.
Операцията * се реализира от функцията IterView.
Сега ще дефинираме клас, който реализира свързаното представяне на списък с две връзки.
Първо ще дефинираме помощна структура Item, която реализира тройната кутия, чрез която се представят елементите на списъка.
За по-голяма общност, дефинираме структурата Item и класа DLList като шаблони.

```
template <class T> struct Item {
	T inf;
	Item *pred;
	Item *succ;
};
```

Сега вече сме готови да дефинираме класът DLList. Тъй като двусвързаният списък се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T> class DLList {
 public:
	DLList(const T&);
	DLList();
	~DLList();
	DLList(const DLList &);
	DLList& operator= (const DLList &);
	void IterStart (Item <T>* = NULL);
	void IterEnd (Item <T>* = NULL);
	Item <T>* IterPred();
	Item <T>* IterSucc();
	bool IterView(T &) const;
	void InsertToEnd (const T&);
	void InsertToBegin (const T&);
	void InsertAfter (Item <T>*, const T&);
```

```
        void InsertBefore (Item <T>*, const T&);
        void DeleteAfter (Item <T>*, T &);
        void DeleteBefore (Item <T>*, T &);
        void DeleteElement (Item <T>*, T &);
        int length() const;
        void print() const;
        void concat (const DLList &);
        void reverse();
        bool empty () const;
private:
        Item<T> *start, *end, *current;
        void delDLList();
         void copyDLList(const DLList &);
};
template <class T> DLList<T>::DLList (const T &x){
        start = new Item <T>;
        start -> inf = x;
        start -> pred = start -> succ = NULL;
        end = start;
}
template <class T> DLList<T>::DLList(){
        start = end = NULL;
}
template <class T> DLList<T>::~DLList(){
        delDLList();
}
template <class T> DLList<T>::DLList (const DLList <T> &r){
        copyDLList(r);
}
template <class T> DLList<T>& DLList<T>::operator= (const DLList<T> &r){
        if (this != &r) {
                delDLList();
                copyDLList(r);
        }
        return *this;
}
template <class T> void DLList<T>::delDLList(){
        Item <T> *p;
        while (start){
                p = start;
                start = start -> succ;
                delete p;
        }
        end = NULL;
}


template <class T> void DLList<T>::copyDLList (const DLList<T> &r) {
        start = end = NULL;
        Item <T> *p = r.start;
        while (p) {
                InsertToBegin(p -> inf);
                p = p -> link;
        }
}
template <class T> void DLList<T>::IterStart (Item <T> *p) {
        if (p) current = p;
        else current = start;
}
template <class T> void DLList<T>::IterEnd (Item <T> *p){
        if (p) current = p;
        else current = end;
}
```

```
template <class T> Item <T> * DLList<T>::IterPred () {
	Item <T> *p = current;
	if (current) current = current -> pred;
	return p;
}
template <class T> Item <T> * DLList<T>::IterSucc (){
	Item <T> *p = current;
	if (current) current = current -> succ;
	return p;
}
template <class T> bool DLList<T>::IterView (T &x) const {
	if (!current) return false;
	x = current -> inf;
	return true;
}
template <class T> void DLList<T>::InsertToEnd (const T x) {
	Item <T> *p = new Item <T>;
	p -> inf = x;
	p -> pred = end;
	p -> succ = NULL;
	if (!start) { start = end = p; }
	else { end -> succ = p; end = p; }
}
template <class T> void DLList<T>::InsertToBegin (const T x){
	Item <T> *p = new Item <T>;
	p -> inf = x;
	p -> pred = NULL;
	p -> succ = start;
	if (!end) { start = end = p; }
	else { start -> pred = p; start = p; }
}
template <class T> void DLList<T>::InsertAfter (Item <T> *p, const T x) {
	Item <T> *q = new Item <T>;
	q -> inf = x;
	q -> pred = p;
	q -> succ = p -> succ;
	p -> succ = q;
	if (q -> succ) q -> succ -> pred = q;
	else end = q;
}
template <class T> void DLList<T>::InsertBefore (Item <T> *p, const T x){
	Item <T> *q = new Item <T>;
	q -> inf = x;
	q -> succ = p;
	q -> pred = p -> pred;
	p -> pred = q;
	if (q -> pred) q -> pred -> succ = q;
	else start = q;
}
template <class T> void DLList<T>::DeleteAfter (Item <T> *p, T &x){
	Item <T> *q = p -> succ;
	x = q -> inf;
	p -> succ = q -> succ;
	if (q -> succ) q -> succ -> pred = p;
	else end = p;
	delete q;
}
template <class T> void DLList<T>::DeleteBefore (Item <T> *p, T &x) {
	Item <T> *q = p -> pred; x = q -> inf;
	p -> pred = q -> pred;
	if (q -> pred) q -> pred -> succ = p;
	else start = p;
	delete q; }
```

```
template <class T> void DLList<T>::DeleteElement (Item  <T> *p, T &x){
        x = p -> inf;
        if (start == end) { start = end = NULL; }
        else if (p == start) {
                start = start -> succ;
                start -> pred = NULL;
        }
        else if (p == end){
                end = end -> pred;
                end -> succ = NULL;
        }
        else {
                p -> pred -> succ = p -> succ;
                p -> succ -> pred = p -> pred;
        }
        delete p;
}
template <class T> int DLList<T>::length () const {
        Item <T> *p = start;
        int n = 0;
        while (p) { n++; p = p -> succ; }
        return n;
}
template <class T> void DLList<T>::print () const {
        Item <T> *p = start;
        while (p) {
                cout << p -> inf << ' ';
                p = p -> succ;
        }
        cout << endl;
}
template <class T> void DLList<T>::concat (const DLList<T> &r) {
        Item <T> *p = r.start;
        while (p) {
                InsertToEnd (p -> inf);
                p = p -> succ;
        }
}
template <class T> void DLList<T>::reverse () {
        Item <T> *p, *temp;
        if (start == end) return;
        p = start;
        start = end;
        end = p;
        while (p) {
                temp = p -> succ;
                p -> succ = p -> pred;
                p -> pred = temp;
                p = temp;
        }
}
template <class T> bool DLList<T>::empty() const {
        return start == NULL;
 }
```

Функциите IterPred, IterSucc, IterStart, IterEnd и IterView реализират т.н. **итератор**. Итераторът е указател към текущ елемент на редица. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:

++ - приложена към итератор, премества този итератор да сочи към следващия елемент на редицата;

-- - приложена към итератор, премества този итератор да сочи към предишния елемент на редицата;

* - приложена към итератор, дава елемента, към който сочи итератора.

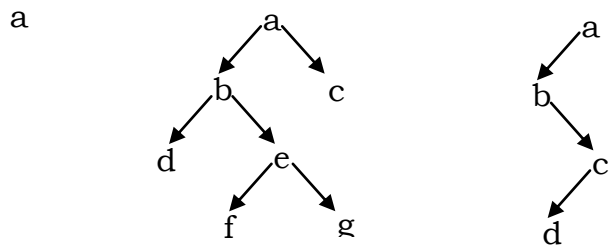
В шаблона DLList, итераторът е current. Функцията IterStart инициализира итератора да сочи към началото на списъка, ако за нея не е указан аргумент или е указан аргумент NULL. Ако за нея е указан друг аргумент, той е началната стойност на итератора. Аналогично, функцията IterEnd инициализира итератора да сочи към края на списъка, ако за нея не е указан

аргумент или е указан аргумент NULL. Ако за нея е указан друг аргумент, той е началната стойност за итератора. Операцията ++ за итератора се реализира от функцията IterSucc, която между другото връща стария итератор. Операцията -- се реализира от функцията IterPred, която между другото връща стария итератор. Операцията * се реализира от функцията IterView.

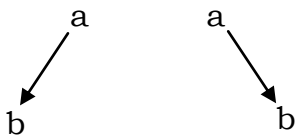
Двоично дърво от тип T е рекурсивна структура от данни, която е или празна или е образувана от:

- данна от тип T, наречена **корен** на двоичното дърво;
- двоично дърво от тип T, наречено **ляво поддърво** на двоичното дърво;
- двоично дърво от тип T, наречено **дясно поддърво** на двоичното дърво.

Множеството на **върховете** (**възлите**) на едно двоично дърво се определя рекурсивно: празното двоично дърво няма върхове, а върховете на непразно дърво са неговият корен и върховете на двете му поддървета.
Ще разгледаме примери. Нека a, b, c, d, e, f и g са данни от тип T. Тогава следните графични представяния определят двоични дървета от тип T.



Посоката на линиите, свързващи върховете с поддървета позволява да се различи ляво от дясно поддърво. Следните две двоични дървета са различни:



В първия случай дясното поддърво е празно, а в другия случай дясното поддърво не е празно.
Листата на двоичното дърво са върховете с две празни поддървета. Например, на първата рисунка листата на трите дървета са a; d, f, g, c; d съответно.
Вътрешни върхове на двоичното дърво са върховете, различни от корена и листата. Например, на първата рисунка първото дърво няма вътрешни върхове, вътрешните върхове на второто дърво са b, e и на третото дърво са b, c.
Ляв наследник на един връх е коренът на лявото му поддърво (ако то е непразно). **Десен наследник** на един връх е коренът на дясното му поддърво (ако то е непразно). Ясно е, че листата на едно дърво са точно онези върхове, които нямат нито ляв, нито десен наследник. Ако a е наследник на b (ляв или десен), казваме, че b е **родител** (**баща**) на a. Ясно е, че всеки връх освен корена има точно един баща.
На всеки връх в дървото може да се съпостави **ниво**. Коренът на дървото има ниво 1 (или 0). Ако един връх има ниво *i*, то неговите наследници имат ниво *i*+1. С други думи, нивото на един връх е броят на върховете, които трябва да бъдат обходени за да се стигне до този връх от корена. Максималното ниво на едно дърво се нарича негова **височина**. Над структурата от данни двоично дърво са възможни следните операции:

- достъп до връх – възможен е пряк достъп до корена и непряк достъп до останалите върхове;
- включване и изключване на връх – възможни са на произволно място в двоичното дърво, резултатът трябва отново да е двоично дърво от същия тип;
- обхождане – това е метод, позволяващ да се осъществи достъп до всеки връх на дървото един единствен път.

Обхождането е рекурсивна процедура, която се осъществява чрез изпълнение на следните три действия, в някакъв фиксиран ред:

- обхождане на корена;
- обхождане на лявото поддърво;
- обхождане на дясното поддърво.

Най-разпространени са **смесеното обхождане** (първо лявото поддърво, после корена и най-накрая дясното поддърво), **низходящото обхождане** (първо корена, после лявото поддърво и най-накрая дясното поддърво) и **възходящо обхождане** (първо лявото поддърво, после дясното поддърво и най-накрая корена).
Основно се използват три начина за представяне на двоично дърво – **свързано**, **верижно** и **чрез списък на бащите**.
- Свързаното представяне се реализира чрез указател към кутия с три полета – информационно, съдържащо стойността на корена и две адресни, съдържащи представянията на лявото и дясното поддърво.
- При верижното представяне се използват три масива – a[N], b[N] и c[N]. Тук N е броят на върховете в дървото, върховете са номерирани от 0 до N-1. Елементът a[i] на масива a съдържа стойността на i-тия връх на дървото. Елементът b[i] на масива b съдържа индекса на левия наследник на i-тия връх (-1, ако той няма ляв наследник), елементът c[i] на масива c

съдържа индекса на десния наследник на i-тия връх (-1, ако той няма десен наследник). Отделно се пази и индекса на корена на двоичното дърво.

- При представянето чрез списък на бащите се използва един масив p[N]. Тук N е броят на върховете в дървото, върховете са номерирани от 0 до N-1. Елементът p[i] е единствения баща на i-тия връх на дървото (-1, ако този връх е коренът).

Сега ще дефинираме клас, който реализира свързаното представяне на двоично дърво.

Първо ще дефинираме помощна структура Node, която реализира тройната кутия, чрез която се представят върховете на дървото. За по-голяма общност, дефинираме структурата Node и класа Tree като шаблони.

```
template <class T> struct Node {
        T inf;
        Node *left;
        Node *right;
};
```

Сега вече сме готови да дефинираме класът Tree. Тъй като двоичното дърво се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T> class Tree {
public:
        Tree(const T&);
        Tree();
        ~Tree();
        Tree(const Tree &);
        Tree& operator= (const Tree &);
        bool empty () const;
        bool RootTree (T&) const;
        bool LeftTree (Tree &) const;
        bool RightTree (Tree &) const;
        void Create();
        void Create3(const T&, const Tree &, const Tree &);
        void PreOrder() const;
        void InOrder() const;
        void PostOrder() const;
private:
        Node <T> *root;
        void DeleteTree (Node<T> *&);
        void CopyTree (Node <T> * &, const Node<T> *);
        void Preord (const Node <T> *);
        void Inord (const Node <T> *);
        void Postord (const Node <T> *);
        void CreateTree (Node <T> *&);
};
template <class T> Tree<T>::Tree (const T& x){
        root = new Node <T>;
        root -> inf = x;
        root -> left = root -> right = NULL;
}
template <class T> Tree<T>::Tree () {
        root = NULL;
}
template <class T> Tree<T>::~Tree () {
        DeleteTree (root);
}
template <class T> Tree<T>::Tree (const Tree<T> &t) {
        CopyTree (root, t.root);
}
template <class T> Tree<T>& Tree<T>::operator= (const Tree<T> &t) {
        if (this != &t) {
                DeleteTree (root);
                CopyTree (root, t.root);
        }
        return *this;
}
```

```
template <class T> void Tree<T>::DeleteTree (Node <T> * &t) {
	if (t) {
		DeleteTree (t -> left);
		DeleteTree (t -> right);
		delete t;
		t = NULL;
	}
}
template <class T> void Tree<T>::CopyTree (Node <T> * &p, const Node <T> *t) {
	if (t) {
		p = new Node <T>;
		p -> inf = t -> inf;
		if (t -> left ) CopyTree (p -> left, t -> left);
		else p -> left = NULL;
		if (t -> right) CopyTree (p -> right, t -> right);
		else p -> right = NULL;
	}
}
template <class T> bool Tree<T>::empty () const {
	return root == NULL;
}
template <class T> bool Tree<T>::RootTree (T &x) const {
	if (!root) return false;
	x = root -> inf;
	return true;
}
template <class T> bool Tree<T>::LeftTree (Tree <T> &t) const {
	if (!root) return false;
	DeleteTree (t.root);
	CopyTree (t.root, root -> left);
	return true;
}
template <class T> bool Tree<T>::RightTree (Tree <T> &t) const {
	if (!root) return false;
	DeleteTree (t.root);
	CopyTree (t.root, root -> right);
	return true;
}
template <class T> void Tree<T>::PreOrder () const {
	Preord (root);
	cout << endl;
}
template <class T> void Tree<T>::InOrder () const {
	 Inord (root);
	cout << endl;
}
template <class T> void Tree<T>::PostOrder () const {
	Postord (root);
	cout << endl;
}
template <class T> void Tree<T>::Preord (const Node <T> *t){  // низходящо
	if (t) {
		cout << t -> inf << ' ';
		Preord (t -> left);
		Preord (t -> right);
	} }
template <class T> void Tree<T>::Inord (const Node <T> *t) { //смесено
	if (t) {
		Inord (t -> left);
		cout << t -> inf << ' ';
		Inord (t -> right);
	}
}
```

```
template <class T> void Tree<T>::Postord (const Node <T> *t) { //възходящо
	if (t) {
		Postord (t -> left);
		Postord (t -> right);
		cout << t -> inf << ' ';
	}
}
template <class T> void Tree<T>::Create3(const T&rt, const Tree<T> &l, const Tree<T> &r) {
	DeleteTree (root);
	root = new Tree<T>;
	root -> inf = rt;
	CopyTree (root -> left, l.root);
	CopyTree (root -> right, r.root);
}
template <class T> void Tree<T>::Create() {
	DeleteTree (root);
	CreateTree (root);
}
template <class T> void Tree<T>::CreateTree (Node <T> * &t) {
	T x; char c;
	cout << "root: ";
	cin >> x;
	t = new Node <T>;
	t -> inf = x;
	cout << "Left tree of: " << x << "  (y/n)? ";
	cin >> c;
	if (c == 'y' || c == 'Y') CreateTree (t -> left); else t -> left = NULL;
	cout << "Right tree of: " << x << "  (y/n)? ";
	cin >> c;
	if (c == 'y' || c == 'Y') CreateTree (t -> right); else t -> right = NULL;
}
```

Предполагаме, че за елементите на типа T е установена линейна наредба.

Двоично наредено дърво от тип T се дефинира рекурсивно по следния начин: празното двоично дърво от тип T е наредено и непразно двоично дърво от тип T е наредено тогава и само тогава, когато всички върхове на лявото му поддърво са по-малки от корена и всички върхове на дясното му поддърво са по-малки от корена и освен това, лявото и дясното му поддърво са двоични наредени дървета от тип T.

Нека t е двоично наредено дърво от тип T. **Включването** на елемента a от тип T в t се осъществява по следния начин:

- ако t е празното дърво, новото двоично наредено дърво е с корен елемента a и празни ляво и дясно поддървета;
- ако t е непразно и a е по-малко от корена му, a се включва в лявото поддърво на t;
- ако t е непразно и a е по-голям от корена му, a се включва в дясното поддърво на t.

Ще използваме този начин за включване на елементи за да създаваме двоичните наредени дървета. В този случай, те притежават следното свойство: смесеното обхождане сортира във възходящ ред елементите от върховете на дървото.

Изтриването на връх елемент a от двоичното наредено дърво t се извършва по следната схема:

- ако коренът на t е по-голям от a, изтриваме a от лявото поддърво на t;
- ако коренът на t е по-малък от a, изтриваме a от дясното поддърво на t;
- ако коренът на t съвпада с a и t има празно ляво поддърво, то t се заменя с дясното си поддърво;
- ако коренът на t съвпада с a и t има празно дясно поддърво, то t се заменя с лявото си поддърво;
- ако коренът на t съвпада с a и t има непразни ляво и дясно поддърво, то се намира най-големият елемент в лявото поддърво (това става като се спуснем по най-десния клон до достигане на връх с празно дясно поддърво), разменя се неговата стойност със стойността на корена (която е a) и въпросният връх се изтрива (той има празно дясно поддърво, така че влизаме в предния случай).

Сега ще дефинираме клас, който реализира свързаното представяне на двоично наредено дърво, в който са включени гореописаните операции. Ще отбележим, че не допускаме в дървото да има върхове с еднакви стойности (това ограничение, обаче, лесно може да се премахне).

Първо ще дефинираме помощна структура Node, която реализира тройната кутия, чрез която се представят върховете на дървото. За по-голяма общност, дефинираме структурата Node и класа BinOrdTree като шаблони.

```
template <class T> struct Node {
	T inf;
	Node *left;
	Node *right;
};
```

Сега вече сме готови да дефинираме класът BinOrdTree. Тъй като двоичното наредено дърво се реализира в динамичната

памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T> class BinOrdTree {
public:
        BinOrdTree (const T&);
        BinOrdTree ();
        ~BinOrdTree ();
        BinOrdTree (const BinOrdTree &);
        BinOrdTree& operator= (const BinOrdTree &);
        bool empty () const;
        bool RootTree (T&) const;
        bool LeftTree (BinOrdTree &) const;
        bool RightTree (BinOrdTree &) const;
        void PrintSorted () const;
        void AddNode (const T&);
        void DeleteNode (const T&);
        void Create();
private:
        Node<T> *root;
        void DeleteTree (Node <T> *&);
        void Del (Node <T> *&, const T&);
        void CopyTree (Node <T> *&, const Node <T>*);
        void Print (const Node <T> *) const;
        void Add (Node <T> *&, const T&);
};
template <class T> BinOrdTree<T>::BinOrdTree (const T& x){
        root = new Node <T>;
        root -> inf = x;
        root -> left = root -> right = NULL;
}
template <class T> BinOrdTree<T>::BinOrdTree () {
        root = NULL;
}
template <class T> BinOrdTree<T>::~BinOrdTree () {
        DeleteTree (root);
}

template <class T> BinOrdTree<T>::BinOrdTree (const BinOrdTree &t){
        CopyTree (root, t.root);
}
template <class T> BinOrdTree<T>& BinOrdTree<T>::operator= (const BinOrdTree &t) {
        if (this != &t) {
                DeleteTree (root);
                CopyTree (root, t.root);
        }
        return *this;
}
template <class T> void BinOrdTree<T>::DeleteTree (Node <T> * &t) {
        if (t) {
                DeleteTree (t -> left);
                DeleteTree (t -> right);
                delete t;
                t = NULL;
        }
}
```

```
template <class T> void BinOrdTree<T>::CopyTree (Node <T> *&p, const Node <T> *t) {
```

```
	if (t) {
		p = new Node <T>;
		p -> inf = t -> inf;
		if (t -> left) CopyTree (p -> left, t -> left);
		else p -> left = NULL;
		if (t -> right) CopyTree (p -> right, t -> right);
		else p -> right = NULL;
	}
}
template <class T> bool BinOrdTree<T>::empty () const {
	return root == NULL;
}
template <class T> bool BinOrdTree<T>::RootTree (T &x) const {
	if (!root) return false;
	x = root -> inf;
	return true;
}
template <class T> bool BinOrdTree<T>::LeftTree (BinOrdTree <T> &t) const {
	if (!root) return false;
	DeleteTree (t.root);
	CopyTree(t.root, root -> left);
	return true;
}
template <class T> bool BinOrdTree<T>::RightTree (BinOrdTree <T> &t) const {
	if (!root) return false;
	DeleteTree (t.root);
	CopyTree(t.root, root -> right);
	return true;
}
template <class T> void BinOrdTree<T>::PrintSorted () const {
	Print (root);
	cout << endl;
}
template <class T> void BinOrdTree<T>::Print (const Node <T> *t) const{
	if (t) {
		Print (t -> left);
		cout << t -> inf << ' ';
		Print (t -> right);
	}
}
template <class T> void BinOrdTree<T>::AddNode (const T& x) {
	Add (root, x);
}

template <class T> void BinOrdTree<T>::Add (Node <T> * &t, const T& x){
	if (!t) {
		t = new Node <T>;
		t -> inf = x;
		t -> left = t -> right = NULL;
	}
	else if (x < t -> inf) Add (t -> left, x);
	else if (x > t -> inf) Add (t -> right, x);
	else cout << "Duplicate cannot be inserted!" << endl;
}
template <class T> void BinOrdTree<T>::DeleteNode (const T x) {
	 Del (root, x);
}
```

```
template <class T> void BinOrdTree<T>::Del (Node <T> * &t, const T& x) {
	if (!t) return;
	if (x < t -> inf) Del (t -> left, x);
	else if (x > t -> inf) Del (t -> right, x);
	else {
		Node <T> *p;
		if (!(t -> left)) { p = t; t = t -> right; delete p; }
		else if (!(t -> right)) { p = t; t = t -> left; delete p; }
		else {
			p = t -> left;
			while (p -> right) p = p -> right;
			t -> inf = p -> inf;
			Del (t -> left, p -> inf);
		}
	}
}
template <class T> void BinOrdTree<T>::Create () {
	DeleteTree (root);
	T x; char c;
	do {
		cout << "Enter element: ";
		cin >> x;
		AddNode (x);
		cout << "Enter more (y/n)? ";
		cin >> c;
	} while (c == 'y' || c == 'Y');
}
```