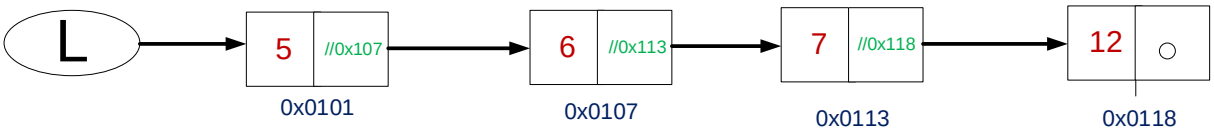
Povezana reprezentacija

Apstraktni tipovi podataka (ATP npr. lista, stek, red, prioritetni red) se obično implementiraju na dva načina, i to:

1. pomoću nizova
2. pomoću povezane reprezentacije.

Za implementaciju ATP-a pomoću povezane reprezentacije koriste se čvorovi koji su međusobno povezani (jednostruko povezani, dvostruko povezani ili sl.).

Grafička ilustracija jednostruko liste koja se sastoji od jednostruko povezanih čvorova je prikazana na slici 1.



Slika 1. Jednostruko povezana lista

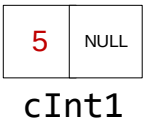
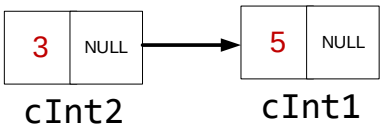
Za implementacija čvora u programskom jeziku C++ koriste se programske strukture (struct) ili programske klase (class).

Cvor.h

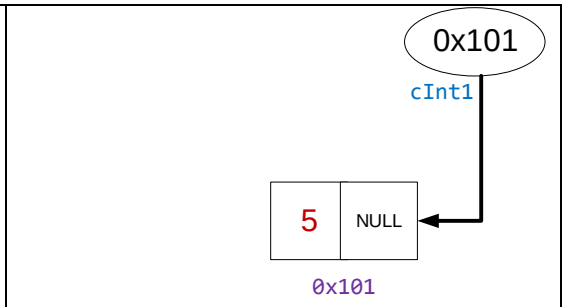
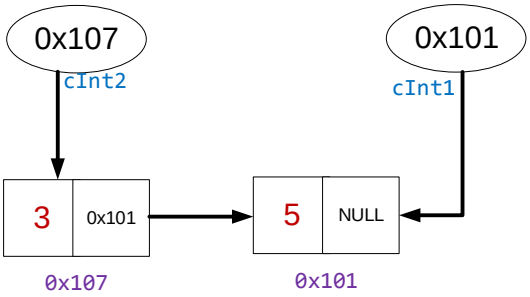
```
#ifndef __CVOR_H__
#define __CVOR_H__

#include <iostream>
using namespace std;

typedef int Tip;

struct Cvor
{
	Cvor* next;
	Tip info;

	Cvor(Tip info, Cvor* next)
	{
		this->info = info;
		this->next = next;
	}
};

#endif
```

Da bi osigurali da čvor (instanca klase ili strukture **Cvor**) nema nepoznate ili beskorisne vrijednosti, dodat je konstruktor koji će prilikom instanciranja objekta zahtijevati stvarne vrijednosti za polje **info** i pokazivač **next**.

Primjer instanciranja objekata tipa **Cvor** u statičkoj memoriji je dat u nastavku.

1	`Cvor cInt1(5, NULL);`	
2	`Cvor cInt2(3, &cInt1);`	

Primjer instanciranja objekata tipa **Cvor** u dinamičkoj memoriji je dat u nastavku.

1	`Cvor* cInt1 = new Cvor(5, NULL);`	
2	`Cvor* cInt2 = new Cvor(3, cInt1);`	

Podatak **info** može da bude bilo kojeg tipa, npr. tipa klasa **Osoba**. Slijedi primjer.

Osoba.h

```
#ifndef __OSOBA_H__
#define __OSOBA_H__

#include <iostream>
using namespace std;

class Osoba
{
public:
	int Id;
	string ime;
	Osoba()
	{
	}
	Osoba(int id, string ime)
	{
		this->ime = ime;
		this->Id = id;
	}
};
#endif
```

Cvor.h

```
#ifndef __CVOR_H__
#define __CVOR_H__

#include "Osoba.h"
typedef Osoba Tip;

struct Cvor
{
	Cvor* next;
	Tip info;

	Cvor(Tip info, Cvor* next)
	{
		this->info = info;
		this->next = next;
	}
};
#endif
```

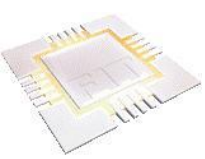
Primjer instanciranja objekata tipa **Cvor** u dinamičkoj memoriji je dat u nastavku.

```
Osoba o1(1623, "M. Hadžić");
Osoba o2(2369, "I. Bešlić");
Osoba o3(2954, "E. Mešić");

Cvor* pOsoba1 = new Cvor(o1, NULL);
Cvor* pOsoba2 = new Cvor(o2, pOsoba1);
Cvor* pOsoba3 = new Cvor(o3, pOsoba2);
```

Međutim, ukoliko je polje info tipa **Osoba** onda je neophodno da klasa **Osoba** ima defaultni konstruktor. Stoga se često koristi **Osoba*** umjesto **Osoba**.

```
#ifndef __CVOR_H__
#define __CVOR_H__

#include "Osoba.h"
typedef Osoba* Tip;

struct Cvor
{
	Cvor* next;
	Tip info;

	Cvor(Tip info, Cvor* next)
	{
		this->info = info;
		this->next = next;
	}
};
#endif
```

```
Osoba* o1 = new Osoba(1623, "M. Hadžić");
Osoba* o2 = new Osoba(2369, "I. Bešlić");
Osoba* o3 = new Osoba(2955, "A. Zelić");

Cvor* pOsobaP1 = new Cvor(o1, NULL);
Cvor* pOsobaP2 = new Cvor(o2, pOsobaP1);
Cvor* pOsobaP3 = new Cvor(o3, pOsobaP2);

Cvor* pOsobaP4 = new Cvor(new Osoba(2955, "A. Zelić"), pOsobaP3);
Cvor* pOsobaP5 = new Cvor(new Osoba(2956, "I. Sudić"), pOsobaP4); //itd.
```



Generičke klase i funkcije

Iz prethodnih primjera je vidljivo da objekat tipa **Cvor** može pamtiti bilo koji tip podatka (int, float, char, Osoba itd.). Međutim, prethodni primjeri imaju ograničenje što unutar jednog C++ programa nije moguće koristiti različite tipove, npr. da varijabla **a** tipa **Cvor** pamti vrijednost *int*, a varijabla **b** tipa **Cvor** pamti podatak *float* i sl. Stoga je u C++ uveden mehanizam koji će omogućiti da se prilikom instanciranja objekata proslijedi i tip podatka. Taj mehanizam se implementira pomoću generičkih funkcija i generičkih klase (ili struktura).

Instanciranje objekata generičkih klasa je prikazano na primjeru čvorova koji pamte **int** i **float**.

```
Cvor<int>* cInt1 = new Cvor<int>(5, NULL);
Cvor<int>* cInt2 = new Cvor<int>(3, cInt1);
Cvor<int>* cInt3 = new Cvor<int>(1, cInt2);

Cvor<float>* pFloat1 = new Cvor<float>(2.4, NULL);
Cvor<float>* pFloat2 = new Cvor<float>(4.2, pFloat1);
Cvor<float>* pFloat3 = new Cvor<float>(5.6, pFloat2);
```

Primjer instanciranja objekata sa generičkim paramterom **Osoba**.

```
Osoba o1(1623, "M. Hadžiæ");
Osoba o2(2369, "I. Bešliæ");
Osoba o3(2954, "E. Mešiæ");

Cvor<Osoba>* cOsoba1 = new Cvor<Osoba>(o1, NULL);
Cvor<Osoba>* cOsoba2 = new Cvor<Osoba>(o2, cOsoba1);
Cvor<Osoba>* cOsoba3 = new Cvor<Osoba>(o3, cOsoba2);
```

Primjer instanciranja objekata sa generičkim paramterom **Osoba***.

```
Osoba* p1 = new Osoba(1623, "M. Hadžiæ");
Osoba* p2 = new Osoba(2369, "I. Bešliæ");
Osoba* p3 = new Osoba(2954, "E. Mešiæ");

Cvor<Osoba*>* cOsobap1 = new Cvor<Osoba*>(p1, NULL);
Cvor<Osoba*>* cOsobap2 = new Cvor<Osoba*>(p2, cOsobap1);
Cvor<Osoba*>* cOsobap3 = new Cvor<Osoba*>(p3, cOsobap2);
```

Generičke klase i strukture se definiraju tako što se naziv generičkog parametra (npr. **Tip**) navede iznad naziva klase ili strukture.

```
#ifndef __CVOR_H__
#define __CVOR_H__

template <class Tip>
struct Cvor
{
        Cvor* next;
        Tip info;

        Cvor(Tip info, Cvor<Tip>* next)
        {
                this->info = info;
                this->next = next;
        }
};
#endif
```

Lista bez tekućeg elementa

U sljedećem primjeru je prikazana Lista bez tekućeg elementa koja je implementirana pomoću povezane reprezentacije.

```
#ifndef __LISTA_H__
#define __LISTA_H__

#include "Cvor.h"

template <class Tip>
class Lista
{
private:
	Cvor<Tip>* prvi;
public:
	Lista()
	{
		prvi = NULL;
	}
	void dodajPrvi(Tip v)
	{
		Cvor<Tip>* t = new Cvor<Tip>(v, prvi);
		prvi = t;
	}

	Tip ukloniPrvi()
	{
		if (IsPrazan())
		{
			throw exception("Greska. Nije moguce ukloniti iz prazne liste");
		}

		Cvor<Tip>* t = prvi;
		prvi = prvi->next;
		Tip x = t->info;
		delete t;
		return x;
	}

	void dodajPosljednji(Tip v)
	{
		Cvor<Tip>* n = new Cvor<Tip>(v, NULL);

		if (prvi == NULL)
		{
			prvi = n;
		}
		else
		{
			// dok t ne bude pokazivao na zadnji èvor
			Cvor<Tip>* t = prvi;
			while (t->next != NULL)
			{
				t = t->next;
			}

			t->next = n;
		}
	}
```

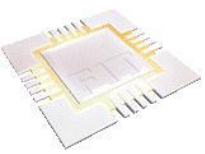
```
        Tip ukloniPosljednji()
        {
                if (IsPrazan())
                {
                        throw exception("Greska. Nije moguce ukloniti iz prazne liste");
                }
                else
                {
                        // dok t ne bude pokazivao na zadnji èvor
                        Cvor<Tip>* t = prvi;
                        Cvor<Tip>* bt = NULL;
                        while (t->next != NULL)
                        {
                                bt = t;
                                t = t->next;
                        }

                        if (bt != NULL)
                        {
                                //ako lista ima više od jednog elementa
                                bt->next = NULL;
                        }
                        else
                        {
                                //ako lista ima jedan elementa
                                prvi = NULL;
                        }

                        Tip rezultat = t->info;
                        delete t;

                        return rezultat;
                }
        }
        bool IsPrazan()
        {
                return (prvi == NULL);
        }
};
#endif
```

Primjer instanciranja liste **x1** u statičkoj i liste **x2** dinamičkoj memoriji je prikazan u nastavku. Iako je objekat tipa **Lista** alociran u statičkoj memoriji, njegovi čvorovi se alociraju u dinamičkoj memoriji (isto kao kod liste u dinamičkoj memoriji).

```
Lista<int> x1;//int

Lista<int>* x2 = new Lista<int>;//int
```

U nastavku je prikazan primjer dodavanja i uklanjanja elemenata iz liste koja pamti podatak tipa **int**.

```
Lista<int> x1;//int
x1.dodajPosljednji(2);
x1.dodajPrvi(3);
cout << "LISTA x1: ukloniPrvi-> " <<x1.ukloniPrvi() << endl;
x1.dodajPrvi(4);
x1.dodajPosljednji(2);
x1.dodajPrvi(6);

while (!x1.IsPrazan())
{
        cout << "LISTA x1: ukloniPosljednji-> " << x1.ukloniPosljednji() << endl;;
}
```



```
Lista<int>* x2 = new Lista<int>;//int
x2->dodajPosljednji(2);
x2->dodajPrvi(3);
cout << "LISTA x2: ukloniPrvi-> " << x2->ukloniPrvi() << endl;
x2->dodajPrvi(4);
x2->dodajPosljednji(2);
x2->dodajPrvi(6);

while (!x2->IsPrazan())
{
	cout << "LISTA x2: ukloniPosljednji-> " << x2->ukloniPosljednji() << endl;
}
```

U nastavku je prikazan primjer dodavanja i uklanjanja elemenata iz liste koja pamti podatak tipa **Osoba**.

```
Osoba o11(11, "o11");
Osoba o12(12, "o12");
Osoba o13(13, "o13");
Osoba o14(14, "o14");
Osoba o15(15, "o15");

Lista<Osoba>* x3 = new Lista<Osoba>;//int

x3->dodajPosljednji(o11);
x3->dodajPrvi(o12);

Osoba oPrvi = x3->ukloniPrvi();
cout << "LISTA x3: ukloniPrvi-> " << oPrvi.Id << "-"<< oPrvi.ime << endl;

x3->dodajPrvi(o13);
x3->dodajPosljednji(o14);
x3->dodajPrvi(o15);

while (!x3->IsPrazan())
{
	Osoba oPoslj = x3->ukloniPosljednji();
	cout << "LISTA x3: ukloniPosljednji-> " << oPoslj.Id << "-" << oPoslj.ime << endl;
}
```

U nastavku je prikazan primjer dodavanja i uklanjanja elemenata iz liste koja pamti podatak tipa **Osoba*****.

```
Lista<Osoba*>* x4 = new Lista<Osoba*>;//int

x4->dodajPosljednji(p21);
x4->dodajPrvi(p22);

Osoba* pPrvi = x4->ukloniPrvi();
cout << "LISTA x4: ukloniPrvi-> " << pPrvi->Id << "-" << pPrvi->ime << endl;

x4->dodajPrvi(p23);
x4->dodajPosljednji(p24);
x4->dodajPrvi(p25);

while (!x4->IsPrazan())
{
	Osoba* pPoslj = x4->ukloniPosljednji();
	cout << "LISTA x4: ukloniPosljedn-> " << pPoslj->Id << "-" << pPoslj->ime << endl;
}
```