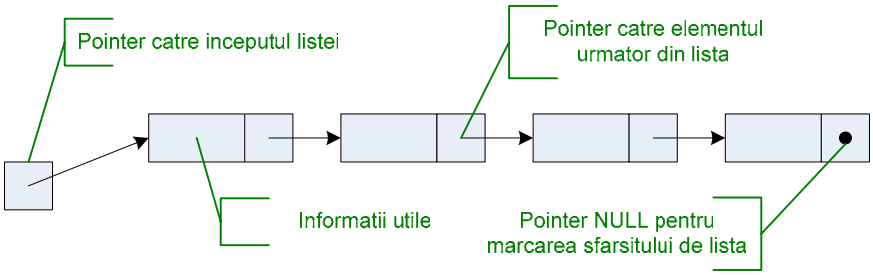
Liste simplu inlantuite

Listele simplu inlantuite sunt structuri de date dinamice omogene. Spre deosebire de masive, listele nu sunt alocate ca blocuri omogene de memorie, ci ca elemente separate de memorie. Fiecare nod al listei contine, in afara ce informatia utila, adresa urmatorului element. Aceasta organizare permite numai acces secvential la elementele listei.

Pentru accesarea listei trebuie cunoscuta adresa primului element (numita capul listei); elementele urmatoare sunt accesate parcurgand lista.

Lista simplu inlantuita poate fi reprezentata grafic astfel:



1. Structura listei

Pentru a asigura un grad mai mare de generalitate listei a fost creat un alias pentru datele utile (in cazul nostru un intreg):

```
// Datele asociate unui
// element dintr-o lista
typedef int Date;
```

In cazul in care se doreste memorarea unui alt tip de date, trebuie schimbata doar declaratia aliasului ***Date***.

Pentru memorarea listei se foloseste o structura autoreferita. Acesta structura va avea forma:

```
// Structura unui element
// dintr-o lista simplu inlantuita
struct Element
{
	// datele efective memorate
	Date valoare;
	// legatura catre nodul urmator
	Element* urmator;
};
```

In cazul in care elemenul este ultimul din lista, pointerul ***urmator*** va avea valoarea NULL.

Declararea listei se face sub forma:

```
// declarare lista vida
Element* cap = NULL;
```

2. Operatii cu liste

Principalele operatii cu liste sunt:

Parcurgere si afisare lista

Lista este parcursa pornind de la pointerul spre primul element si avansand folosind pointerii din structura pana la sfarsitul listei (pointer NULL).

```
// Parcurgere si afisare lista simpla
void Afisare(Element* cap)
{
	// cat timp mai avem elemente
	// in lista
	while (cap != NULL)
	{
		// afiseaza elementul curent
		cout << cap->valoare << endl;

		// avanseaza la elementul urmator
		cap = cap->urmator;
	}
}
```

Inserare element

Inserarea unui element se poate face la inceputul sau la sfarsitul listei.

a) Inserare la inceput
Acesta este cazul cel mai simplu: trebuie doar alocat elementul, legat de primul element din lista si repozitionarea capului listei:

```
// Inserare element la inceputul unei
// liste simplu inlantuite
void InserareInceput(Element* &cap, Date val)
{
	// Alocare nod si initializare valoare
	Element *elem = new Element;
	elem->valoare = val;

	// legare nod in lista
	elem->urmator = cap;

	// mutarea capului listei
	cap = elem;
}
```

b) Inserare la sfarsitul listei

In acest caz trebuie intai parcursa lista si dupa aceea adaugat elementul si legat de restul listei. De asemenea, trebuie avut in vedere cazul in care lista este vida.

```
// Inserare element la sfarsitul unei
// liste simplu inlantuite
void InserareSfarsit(Element* &cap, Date val)
{
```

```
	// Alocare si initializare nod
	Element *elem = new Element;
	elem->valoare = val;
	elem->urmator = NULL;

	// daca avem lista vida
	if (cap == NULL)
		// doar modificam capul listei
		cap = elem;
	else
	{
		// parcurgem lista pana la ultimul nod
		Element *nod = cap;
		while (nod->urmator != NULL)
			nod = nod->urmator;

		// adaugam elementul nou in lista
		nod->urmator = elem;
	}
}
```

c) inserare dupa un element dat

```
void InserareInterior(Element* &cap, Element* p, Date val)
{
	// Alocare si initializare nod
	Element *elem = new Element;
	elem->valoare = val;
	elem->urmator = NULL;

    // lista vida
	if (cap == NULL)
	{
		cap = elem;
		return;
	}

	// inserare la inceputul listei
	if (cap == p)
	{
		elem->urmator = cap;
		cap = elem;
		return;
	}

	// inserare in interior
	elem->urmator = p->urmator;
	p->urmator = elem;
}
```

Cautare element

Cautarea unui element dintr-o lista presupune parcurgerea listei pentru identificarea nodului in functie de un criteriu. Cele mai uzuale criterii sunt cele legate de pozitia in cadrul listei si de informatiile utile continute de nod. Rezultatul operatiei este adresa primului element gasit sau NULL.

a) Cautarea dupa pozitie

Se avanseaza pointerul cu numarul de pozitii specificat:

```
// Cautare element dupa pozitie
Element* CautarePozitie(Element* cap, int pozitie)
{
	int i = 0;  // pozitia curenta

	// parcurge lista pana la
	// pozitia ceruta sau pana la
	// sfarsitul listei
	while (cap != NULL && i < pozitie)
	{
		cap = cap->urmator;
		i++;
	}

	// daca lista contine elementul
	if (i == pozitie)
		return cap;
	else
		return NULL;
}
```

b) Cautarea dupa valoare

Se parcurge lista pana la epuizarea acesteia sau identificarea elementului:

```
// Cautare element dupa valoare
Element* CautareValoare(Element* cap, Date val)
{
	// parcurge lista pana la gasirea
	// elementului sau epuizarea listei
	while (cap != NULL && cap->valoare != val)
		cap = cap->urmator;

	return cap;
}
```

Stergere element

a) Stergerea unui element din interiorul listei (diferit de capul listei)

In acest caz avem nevoie de adresa predecesorului elementului de sters. Se modifica legaturile in sensul scurtcircuitarii elementului de sters, dupa care se elibereaza memoria corespunzatoare elementului de sters:

```
// sterge un element din interiorul listei
// primind ca parametru adresa predecesorului
void StergereElementInterior(Element* predecesor)
{
	// salvam referinta la elementul de sters
	Element* deSters = predecesor->urmator;

	// scurcircuitam elementul
	predecesor->urmator = predecesor->urmator->urmator;

	// si il stergem
	delete deSters;
```

```
}
```

b) Stergerea unui element de pe o anumita pozitie

Daca elementul este primul din lista, atunci se modifica capul listei, altfel se cauta elemental si se sterge folosind functia definite anterior:

```
void StergerePozitie(Element* &cap, int pozitie)
{
	// daca lista e vida nu facem nimic
	if (cap == NULL)
		return;

	// daca este primul element, atunci
	// il stergem si mutam capul
	if (pozitie == 0)
	{
		Element* deSters = cap;
		cap = cap->urmator;
		delete deSters;
		return;
	}

	// daca este in interor, atunci folosim
	// functia de stergere
	Element* predecesor = CautarePozitie(cap, pozitie-1);
	StergereElementInterior(predecesor);
}
```

c) stergerea dupa o valoare

Se cauta predecesorul elementului si se foloseste functia de stergere element:

```
void StergereValoare(Element* &cap, Date val)
{
	// daca lista e vida nu facem nimic
	if (cap == NULL)
		return;

	// daca este primul element, atunci
	// il stergem si mutam capul
	if (cap->valoare == val)
	{
		Element* deSters = cap;
		cap = cap->urmator;
		delete deSters;
		return;
	}

	// cautam predecesorul
	Element* elem = cap;
	while (elem->urmator != NULL && elem->urmator->valoare != val)
		elem = elem->urmator;


	// daca a fost gasit, atunci il stergem
	if (elem->urmator != NULL)
		StergereElementInterior(elem);
}
```

3. Probleme

Sa se realizeze functiile pentru:
1. Interschimbarea a doua elemente prin modificarea legaturilor.
2. Concatenarea a doua liste simplu inlantuite.
3. Copierea unei liste simplu inlantuite.
4. Stergerea unei liste simplu inlantuite.
5. Sortarea unei liste simplu inlantuita.
6. Gasirea elementului aflat pe pozitia *i* de la sfarsitul listei.
7. Inversarea elementelor unei liste prin modificarea legaturilor.
8. Stabilirea simetriei unei liste.
9. Conversia unei matrice in matrice rara memorata ca lista simpla.

10. Se considera o lista ce contine elemente care au ca informatie utila: cod produs, cantitate si pret. Scrieti si apelati functia care calculeaza total valoare pentru materialele existente in lista.

11. Enumerati avantajele si dezavantajele listelor simple fata de masive alocate static/dinamic. Dati exemple de cazuri in care este mai eficenta flosirea listelor si cazuri in care este mai eficienta folosirea vectorilor.