Упражнение 9 – Едносвързани списъци

Понятие за списък

В компютърните системи събирането на едно място на данни от един и същи тип се нарича колекция от данни. Колекциите са много разнообразни като видове – масиви, списъци, дървета и т.н.

До сега изучаваните масиви дори са само частен случай на колекция, която както е известно се намира на едно място в оперативната памет, а достъпът се осъществява посредством индекс (отстояние от началния елемент).

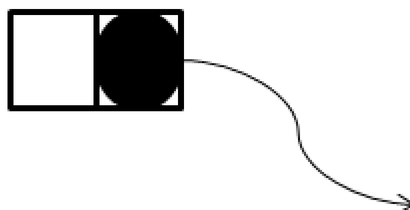
Съществува вид колекция, при който елементите не са един до друг, а за разпръснати из паметта и всеки елемент знае къде се намира следващия след него. Тази колекция се нарича едносвързан списък. В този вид списък един елемент „знае“ адреса на следващия елемент. Така става възможно ангажирането на големи по обем памет, без да е завължително изискване тя да се намира блоково само на едно място. Също така, посредством списъци някои от операциите за работа с колекции се улесняват – например изтриване на елемент, вмъкване на елемент.

Дефиниране на едносвързан списък

Списъкът представлява структура от данни, като тази структура е разделена на две части – част за потребителски данни и указател към следващ елемент.

```
/*Примерно съдържание на списъчна структура*/
struct List
{
	int val;
	struct List* next;
};
typedef struct List List;
```

На примера по-горе е показан елемент от списък, който има поле за потребителски данни – цяло число (тук, то може да бъде променяно за целите на реализираната програма), както и от указател към същата структура. Тук възниква въпрос, дали този указател няма да причини зацикляне? Първо, както беше разгледано на предишни упражнения, указателят представлява число, а не цялата структура, следователно това само „ще посочва“ (ще съдържа адреса на следващия елемент, но няма да е следващия елемент). Така реализивана структурата позволява безкраен брой елементи да се добавят към списъка.



хон.ас. инж. Боян Петров, ***b_petrov@tu-sofia.bg***,
София, 2013

Употреба на списък

За правилната употреба на списък за необходими две съставки: Указатели към началото и указател към текущ елемент. Нуждата от две променливи се изразява във това, че не съществува елемент от списъка, който да сочи към първият елемент - корен, за това трябва неговото местоположение да се пази, точно поради тази причина се налага дефинирането на указател от списък, а като стойност ще му бъде подаден адрес на съществуващия елемент. Другата променлива има за цел да обслужва текущ елемент от списъка.

```
List *root;
List *curr_item;
```

Създаване на нов елемент

Създаването на нов елемент е необходим, защото данните тук са динамични и достъпът до тях е през указатели, а не директен, следователно наличието на указатели в програмата не означава, че елеметите съществуват, за това първа стъпка е създаването на нов елемент.

```
/*Създаване на нов елемент
*/
List *new_item()
{
	List *result=NULL;
	result = (List *)malloc(sizeof(List));
	printf("New Item -> Val = ");
	scanf("%d",&result->val);
	result->next = NULL;
	return result;
}
```

В примерът е показан как се създава един елемент, чрез функция. Функцията връща адреса на вече създадения елемент. В кода се забелязва, че потребителските стойности се въвеждат от клавиатурата, но тук това действие може да бъде модифицирано като стойностите се въвеждат като параметри на функцията.

При първоначално създаване на списък е необходимо адресът на първия елемент да се присвои на корена (началото) на списъка, след това се изграждат и следващите елементи.

Обхождане на списък

Обхождането на списък се различава съществено от обхождането на масив. Тук идеята е „да се взима следващия елемент, докато има следващ". За критерий дали има или няма следващ се съди по стойността на указателя next, елемент на структурата List, ако next на елемент от списъка е NULL, то той няма следващ елемент. Пример:

```
List* root;
List* curr_item;

curr_item = root;
while(curr_item != NULL)
{
	// Работа с текущ елемент
	curr_item=curr_item->next; // Отиване при следващ елемент
}
```

Изграждане на списък

Съществуват два основни метода за изграждане на списък – стеков или опашъчен.

Стеков списък

При стековия списък новите елементи се добавят в началото на списъка. Както е показано в примера:

```
/*Създаване на стеков списък*/
List *create_stack_list(int size)
{
	List *res_root = NULL;
	List *curr_item = NULL;
	int i;
	for(i=0; i<size; i++)
	{
		curr_item = new_item();
		curr_item->next = res_root;
		res_root=curr_item;
	}
	return res_root;
}
```

Опашка

При опашката новите елементи в списъка се добавят след последния елемент:

```
/*Създаване на опашъчен списък*/
List *create_queue_list(int size)
{
	List *res_root = NULL;
	List *end_item = NULL;
	int i;
	for(i=0; i<size; i++)
	{
		if(res_root == NULL)
		{
			res_root = new_item();
			end_item = res_root;
		}
		else
		{
			end_item->next = new_item();
			end_item = end_item->next;
		}
	}
	return res_root;
}
```

Резултатът от двете примерни функции е нов списък, където са разположени всички създадени елементи.

Търсене в списък

Търсенето в списък е сравнително лесно. Като алгоритъм: започва се от първия елемент и ако текущия елемент отговаря на дадено условие, то даденият елемент е намерен.

```
/*Връща указател към първият елемент от списък, който отговаря на зададен критерии*/
List* find_first_item(List* root, int criteria(List *item))
{
	List* curr_item = root;
```

```
        while(curr_item != NULL)
        {
                if(criteria(curr_item))
                        return curr_item;
                curr_item = curr_item->next;
        }
        return NULL;
}
```

В текущият пример критерият е зададен като външна функция, която се подава като параметър на фунцията, а резултатът е първият елемент от списъка, който отговаря на условието.

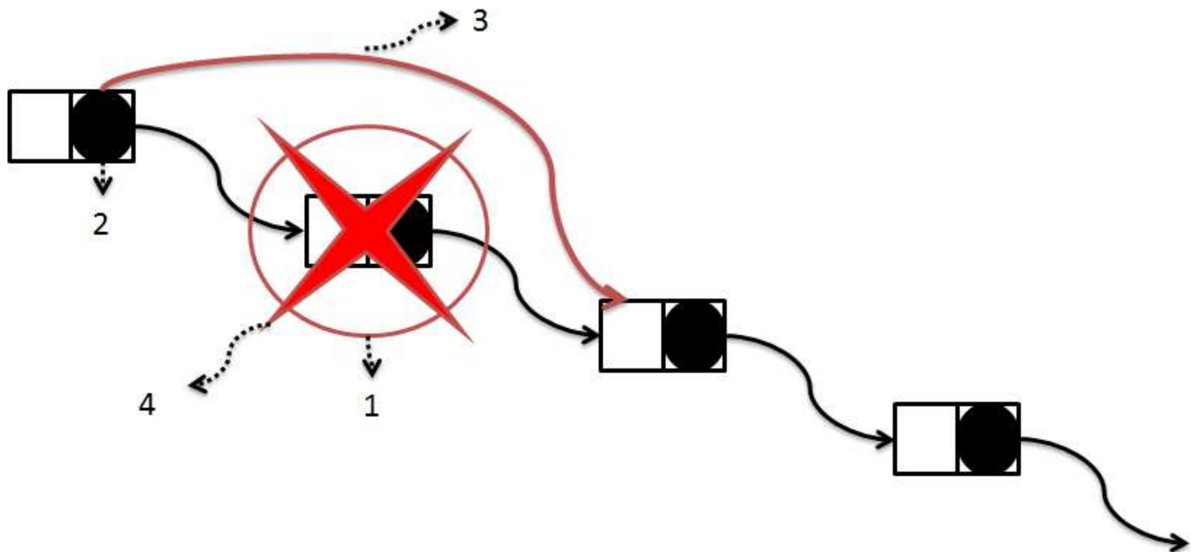
Изтриване на елемент

Изтриването на елемент се състои от няколко стъпки:

1. откриване на елемента за изтриване;
2. откриване на елемента преди елемента за изтриване (предходния);
3. Предходен елемент->next става равно на адресът на елемент за изтриване->next;
4. Изчистване на заетата памет от елемента, който трябва да бъде изтрит.

Идеята тук е да се прекратят всички връзки от и към елемета за изтриване, защото след освобождаването на паметта, няма къде другаде да се запази адресът на елемента, който той сочи.

Стъпките са показани по-нагледно тук, като с пунктир са посочени стъпките:



Примерен код за изтриване на всички елементи по даден критерии:

```
/*Изтриване на всички елемети на списък, които отговарят на зададен критерии*/
List *delete_item(List *root, int criteria(List *item))
{
        List* prev_item=root;
        List* curr_item=root;
        int f=0;
        while(curr_item!=NULL)
        {
```

```
            if(criteria(curr_item))
            {
                  if(curr_item == root)
                  {
                        root = root->next;
                        prev_item = root;
                        f=1;
                  }
                  else
                        prev_item->next = curr_item->next;
                  free(curr_item);
                  curr_item = prev_item;
            }
            prev_item = curr_item;
            if(f == 0)
                  curr_item=curr_item->next;
            f=0;
      }
      return root;
}
```

Освобождаване на памет

Тъй като елементите на списъка са разпръснати, то и освобождаването на памет ще е по-сложен процес. Трябва да бъде освободена паметта, заемана от всяка клетка поотделно, за целта се използва цикъл, който зе върти до изчерване на всички елементи от дадения списък.

```
void free_list(List* root)
{
      List* curr_item = root;
      while(root != NULL)
      {
            root = root->next;
            free(curr_item);
            curr_item= root;
      }
}
```

Примерен код за работа с едносвързани списъци

В примерния код по-долу са показани не само функциите, разгледани по-горе, но и фукции за размяна на два елемента, за разпечатване на всички елементи от даден списък, сортиране по „метода на мехурчето“ и др.

```
/*
(c) Боян Петров, 2012-2013
Технически Университет - София
Факултет "Компютърни системи и управление"
*/

/*
Примерна реализация на функции за обработка на едносвързан списък
*/
#include <stdio.h>
#include <stdlib.h>
/*Примерно съдържание на списъчна структура*/
struct List
{
	int val;
	struct List* next;
};
typedef struct List List;
/*Създаване на нов елемент
*/
List *new_item()
{
	List *result=NULL;
	result = (List *)malloc(sizeof(List));
	printf("New Item -> Val = ");
	scanf("%d",&result->val);
	result->next = NULL;
	return result;
}
/*Създаване на опашъчен списък*/
List *create_queue_list(int size)
{
	List *res_root = NULL;
	List *end_item = NULL;
	int i;
	for(i=0; i<size; i++)
	{
		if(res_root == NULL)
		{
			res_root = new_item();
			end_item = res_root;
		}
		else
		{
			end_item->next = new_item();
			end_item = end_item->next;
		}
	}
	return res_root;
}
/*Създаване на стеков списък*/
List *create_stack_list(int size)
{
	List *res_root = NULL;
	List *curr_item = NULL;
	int i;
```

```
	for(i=0; i<size; i++)
	{
		curr_item = new_item();
		curr_item->next = res_root;
		res_root=curr_item;
	}
	return res_root;
}
/*Изчистване на списък от паметта*/
void free_list(List* root)
{
	List* curr_item = root;
	while(root != NULL)
	{
		root = root->next;
		free(curr_item);
		curr_item= root;
	}
}
/*Изтриване на всички елеmenti на списък, които отговарят на зададен критерии*/
List *delete_item(List *root, int criteria(List *item))
{
	List* prev_item=root;
	List* curr_item=root;
	int f=0;
	while(curr_item!=NULL)
	{
		if(criteria(curr_item))
		{
			if(curr_item == root)
			{
				root = root->next;
				prev_item = root;
				f=1;
			}
			else
				prev_item->next = curr_item->next;
			free(curr_item);
			curr_item = prev_item;
		}
		prev_item = curr_item;
		if(f == 0)
			curr_item=curr_item->next;
		f=0;
	}
	return root;
}
/*Добавя елемент след даден елемент*/
void insert_item_after(List *item,List *item2)
{
	item2->next=item->next;
	item->next = item2;
}
/*Разпечатва на екрана всички елементи от списък*/
void print(List* root)
{
	List* curr_item = root;
	while(curr_item != NULL)
	{
		printf("Item has value %d\n",curr_item->val);
		curr_item=curr_item->next;
	}
```

```
}
/*Връща указател към първият елемент от списък, който отговаря на зададен критерии*/
List* find_first_item(List* root, int criteria(List *item))
{
	List* curr_item = root;
	while(curr_item != NULL)
	{
		if(criteria(curr_item))
			return curr_item;
		curr_item = curr_item->next;
	}
	return NULL;
}
/*Разменя два елемента от един списък*/
List* swap_items(List* root, List* A, List* B)
{
	List *prev_A=NULL;
	List *prev_B=NULL;
	List *curr_item = root;
	int f=0;
	if(root == A) f=1;
	else if(root == B) f=2;
	while(curr_item != NULL)
	{
		if(curr_item->next == A)
			prev_A = curr_item;
		if(curr_item->next == B)
			prev_B = curr_item;
		curr_item = curr_item->next;
	}
	if((prev_A != NULL||f==1) && (prev_B != NULL||f==2))
	{
		if(f!=1)
			prev_A->next = B;
		else
			root = B;
		if(f!=2)
			prev_B->next = A;
		else
			root = A;
		curr_item = A->next;
		A->next = B->next;
		B->next = curr_item;

	}
	return root;

}
/*Импементация на сортиране по метода на мехурчето върху списък*/
List* bubble_sort(List* root,int compare(List* A, List* B))
{
	List* curr_item = root;
	int change;
	do
		for(curr_item = root,change = 0;curr_item-
>next!=NULL&&curr_item!=NULL;curr_item=curr_item->next)
			if(compare(curr_item,curr_item->next))
			{
				root = swap_items(root,curr_item,curr_item->next);
				change = 1;
				break;
			}
```

```
	while(change);
	return root;
}
/*Функции за определяне на критерии и сравнение*/
int cmp(List* A, List* B)
{
	if(A->val > B->val)
		return 1;
	else
		 return 0;
}
int val_more_than_5_and_less_than_8(List *item)
{
	if(item->val > 5 && item->val < 8)
		return 1;
	else
		return 0;
}
int main()
{
	List *root = NULL;
	int n;
	scanf("%d",&n);
	root=create_queue_list(n);
	printf("Before Sort\n");
	print(root);
	root = bubble_sort(root,cmp);
	printf("After Sort\n");
	print(root);
	free_list(root);
	return 0;
}
```