

Абстрактный тип данных СПИСОК

Операции над абстрактным Списком

- ***CreateList(List)*** - создает пустой список *List*
- ***DeleteList(List)*** – уничтожает список *List*
- ***IsEmpty(List)*** – определяет пуст ли список *List*
- ***Insert(index, NewElement, List)*** - вставляет новый элемент *NewElement* в список *List* на позицию *index*
- ***Remove(index, List)*** – удаляет элемент списка, находящийся в позиции *index*

Операции над абстрактным Списком

- *TypeItem* **Retrive**(*index*, *List*) – возвращает элемент, находящийся в позиции *index*
- **Getlength**(*List*) – возвращает количество элементов в списке *List*
- *Pos* **Find**(*List*, *Element*)- возвращает позицию элемента *Element*
(*Pos* может быть как номером элемента, так и указателем на некоторый элемент)

Реализация списков

- Необходимо определить тип элементов и понятия «позиция» элемента:
- ***typedef int TypeItem*** – тип элемента может быть как простым, так и сложным
- ***typedef int Pos*** – в данном случае позицией элемента будет его номер в списке

Реализация списков посредством массивов

- При реализации с помощью массивов все элементы списка располагаются в смежных ячейках, причем у каждого элемента определен номер.
- Это позволяет легко просматривать список, вставлять и удалять элементы в начало и в конец списка.
- Однако, вставка элемента в середину списка потребует от нас сдвинуть все остальные элементы, также как удаление

Реализация списков посредством массивов

- Определяем максимальное количество элементов:
- `define max_list 100; //`
максимальное число элементов списка

Реализация списков посредством массивов

- Описываем структуру List:

```
Struct List {  
    TypeItem Items [Max_list];  
    //массив элементов списка  
    int last; //индекс следующего  
    элемента  
}
```

Реализация списков посредством массивов

```
Void CreateList(List L)  
{ L.last=0;}
```



```
Viod Insert(int n,TypeItem NewItem,List L)
{
if (L.last>=100) cout<<'Очередь полна';
else
if (n>L.last || n<1)
    cout<<'Такой позиции нет';
else
    {for (i=L.Last; i>=n; i--)
L.Items[i+1]=L.Items[i];
    L.last=L.last+1;
    L.Items[n]=NewItem; }

} //end Insert
```

Viod ***Remove***(int n, List L)

{
 if (n>L.last || n<1)

cout<<'Такой позиции нет';

else

 {L.last=L.last-1;

for (i=n; i<=L.last; i++)

 L.Items[i]=L.Items[i+1];

 }

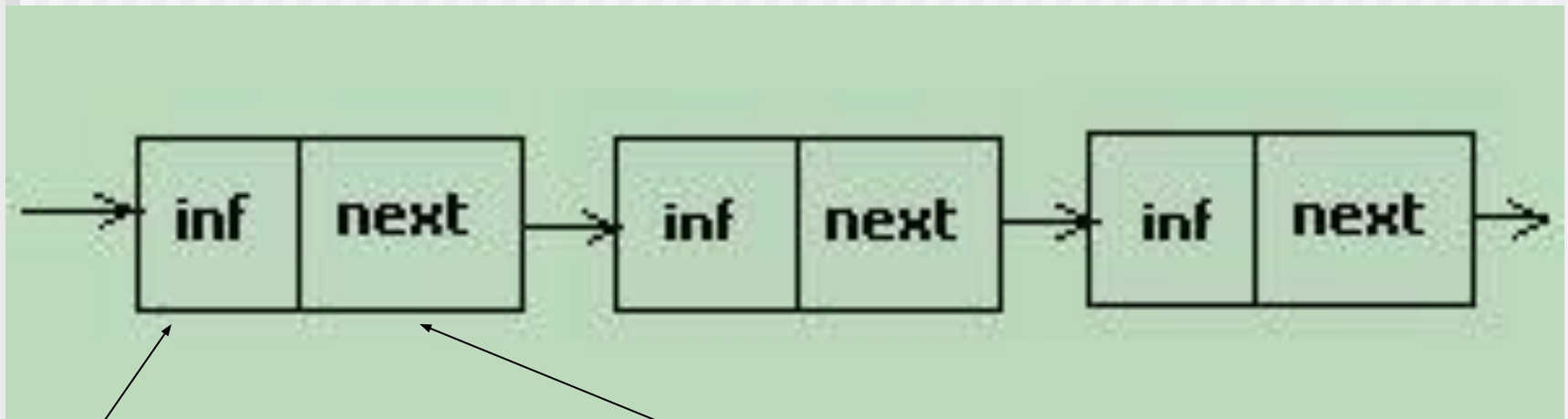
} *//end Remove*

```
Pos Find(TypeItem x, List L)
{for (i=n; i<=L.last; i++)
    if (L.Items[i]=x)
        return(i);
    return(L.last+1); //x не найден
} //end Remove
```

Реализация списков с помощью указателей

- В данном случае элементы списка не обязательно расположены в смежных ячейках, для связывания элементов используются указатели.
- Эта реализация освобождает нас с одной стороны от использования непрерывной области памяти
- Нет необходимости перемещения элементов при вставке или удалении элемента в список.
- Необходима дополнительная память для хранения указателей.

Реализация связанных списков с помощью указателей



информационная
часть

ссылочная часть –
указатель на
следующий элемент

Определение структуры List:

```
typedef struct celltype
{
    TypeItem Item; // элемент списка
    celltype *Next; // указатель на
    следующий элемент
}
typedef celltype *list; //
```

Описания необходимых типов и переменных

- `typedef int Pos;` // позицией элемента будет его номер в списке
- `typedef celltype *Pos;` // позицией элемента будет указатель на этот элемент

Функции работы со списком

Void CreateList(List S) // создание
пустого списка

```
{ S=new celltype;  
  S->next=NULL;  
}
```


void Insert (TypeItem x, Pos n, list S)

{list temp, current;

~~temp=S; current=S->Next;~~

Pos i=1;

while(current!=0)

{if (i==n)

{temp=new celltype;

temp->Next=current->Next;

temp->Item=x;

current->Next=temp;

break;}

```
    i++;  
    current=current->next;  
} //end while
```

```
} //end of insert
```

void Remove (Pos n, list S)

{list current=S->Next, temp;

Pos i=1;

while(current!=NULL && i<n)

{ current=current->next;i++;)

if(i==n){

temp=current->next;

current->next=current->next->next;

delete temp;}

}//end

Pos Find (TypeItem x, list S)

```
{list temp;
```

```
    Pos i=1;
```

```
    if (S->Next==NULL) cout<<'List Null';
```

```
    else {
```

```
        temp=S->Next;
```

```
        while(temp->Next!=NULL)
```

```
            {if (temp->Item==x) return (i);
```

```
              temp=temp->next;i++;}
```

```
    return (0);}
```

```
}//end
```

TypeItem Retrive (Pos n, list S)

```
{list temp;
```

```
    Pos i=1;
```

```
if (S->Next==NULL) cout<<'List Null';
```

```
else {
```

```
temp=S->Next;
```

```
while(temp->Next!=NULL)
```

```
    {if (i==n) return (temp->Item);
```

```
        temp=temp->next;i++;}
```

```
return (0);}
```

```
//end
```

TypeItem Retrive (Pos n, list S)

```
{list temp;
```

```
    Pos i=1;
```

```
if (S->Next==NULL) cout<<'List Null';
```

```
else {
```

```
temp=S->Next;
```

```
while(temp->Next!=NULL)
```

```
    {if (i==n) return (temp->Item);
```

```
        temp=temp->next;i++;}
```

```
return (0);}
```

```
//end
```

Сравнение реализаций

- Реализация списков с помощью массивов требует указания максимального размера массива до начала выполнения программы
- Если длина списка заранее не известна, более рациональным способом будет реализация с помощью указателей.
- Процедуры INSERT и DELETE в случае связанных списков выполняются за конечное число шагов для списков любой длины.

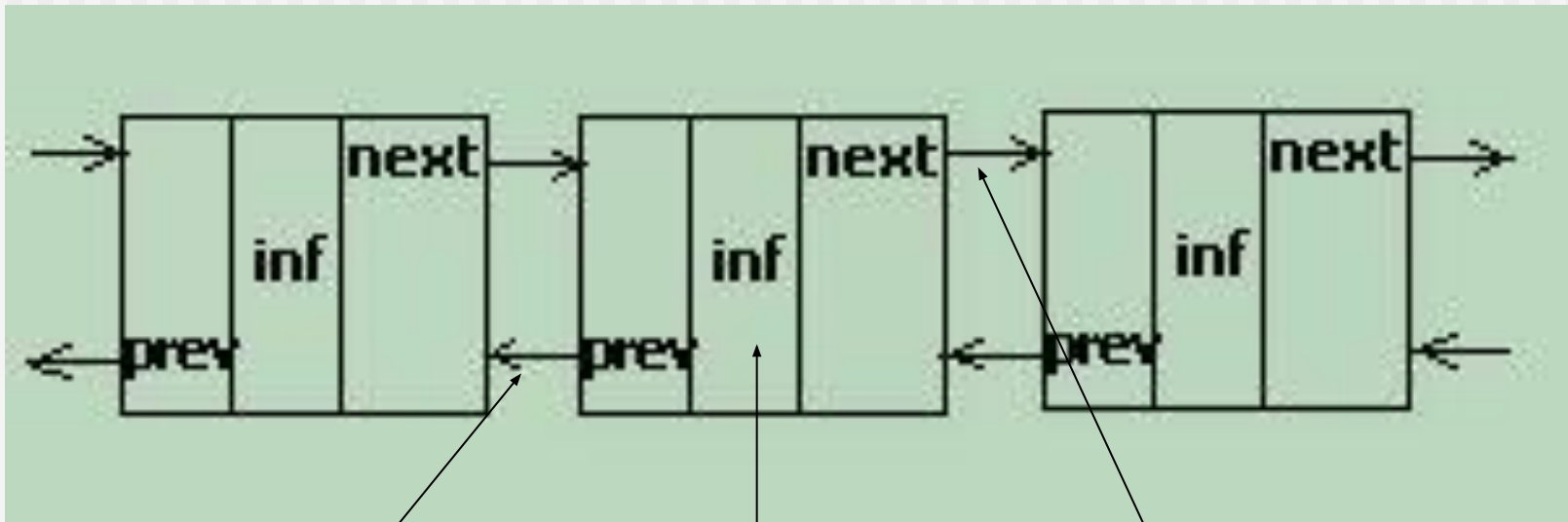
Сравнение реализаций

- Реализация списков с помощью массивов расточительна с точки зрения использования памяти, которая резервируется сразу.
- При использовании указателей необходимо место в памяти для них тоже.
- При использовании указателей нужно работать очень аккуратно. Поэтому в различных случаях бывают более выгодны одни или другие реализации.

Двусвязные списки

- Используются в приложениях, где необходимо организовать эффективное перемещение по списку как прямом, так и в обратном направлениях

Двусвязные списки



информационная
часть

указатель на
предыдущий
элемент

указатель на
следующий
элемент

Описание структуры списка

```
typedef struct celltype
{
    TypeItem Item; // элемент списка
    celltype *Next; // указатель на
                    // следующий элемент
    celltype *Previous; // указатель на
                       // предыдущий элемент
}
typedef celltype *list; //
```