Свързан списък

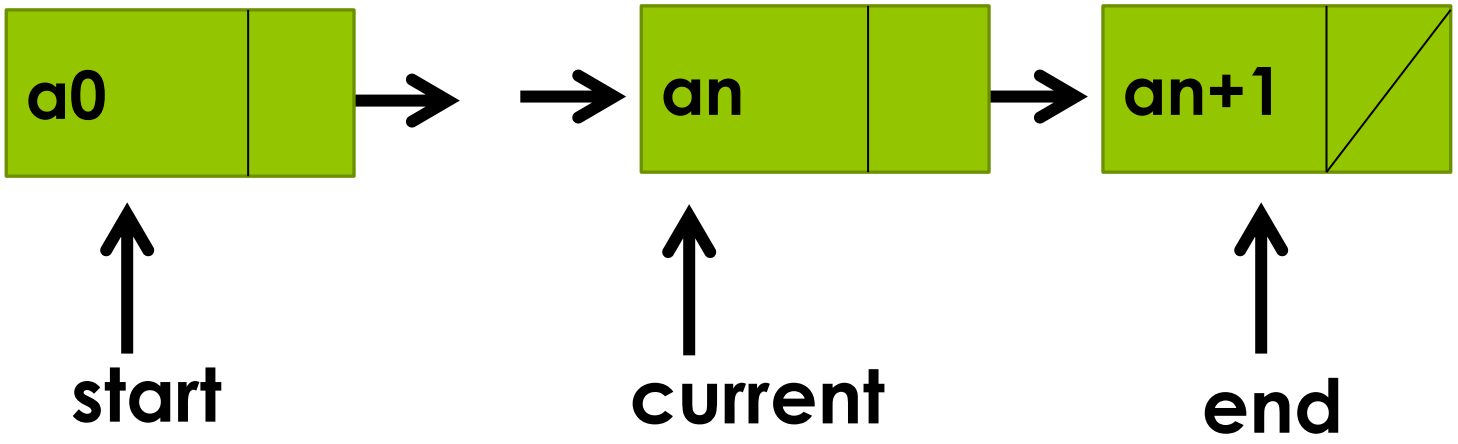
Изготвил:
гл.ас. д-р Нора Ангелова

Свързан списък

- Хомогенна линейна структура от данни
- Възможен е пряк достъп до елемента в единия край на редицата – „начало“ на списъка
- Възможен е последователен достъп до всеки от останалите елементи

Свързан списък с една връзка

- Представяне, аналогично на свързаното представяне на стек и опашка
- Въвеждат се и указатели към края и към текущ елемент на списъка



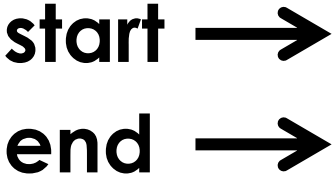
Свързан списък с една връзка

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

```
template <class T>
class LList
{
private:
  elem_link<T> *start;
  elem_link<T> *end;
  elem_link<T> *current;

  void DeleteList();
  void CopyList(LList<T> const &);
```

```
public:
  LList();
  LList(LList<T> const &);
  LList& operator= (LList<T> const &);
  ~LList();

  void IterStart(elem_link<T> *p = NULL);
  elem_link<T>* Iter();

  void ToEnd(T const &);
  void InsertAfter(elem_link<T>*, T const &);
  void InsertBefore(elem_link<T>*, T const &);
  bool DeleteAfter(elem_link<T>*, T &);
  bool DeleteBefore(elem_link<T>*, T &);
  void DeleteElem(elem_link<T>*, T &);
  void print();
  int length();
  void concat(LList<T> const& list);
};
```

```
///
/// Creates an empty list
///
template <class T>
LList<T>::LList()
{
  start = NULL;
  end = NULL;
}
```

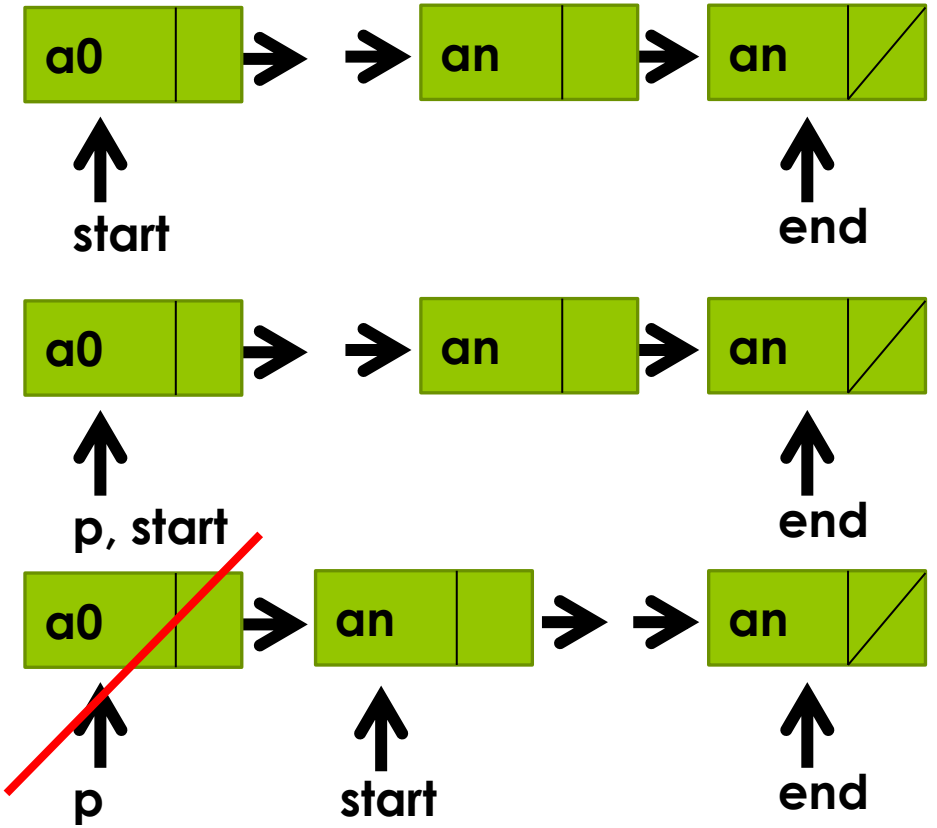


```
///
/// Destroys a list
///
template <class T>
LList<T>::~LList()
{
  DeleteList();
}
```

```
///
/// Copy constructor
///
template <class T>
LList<T>::LList(LList<T> const & list)
{
  CopyList(list)
}
```

```
///
/// Copies the contents of one list to another
///
template <class T>
LList<T>& LList<T>::operator=(LList<T> const & list)
{
  if(this != &list)
  {
    DeleteList();
    CopyList(list);
  }
  return *this;
}
```

```
///
/// Removes all elements of a list
///
template <class T>
void LList<T>::DeleteList()
{
  elem_link<T> *p;
  while(start) {
    p = start;
    start = start->link;
    delete p;
  }

  end = NULL;
}
```



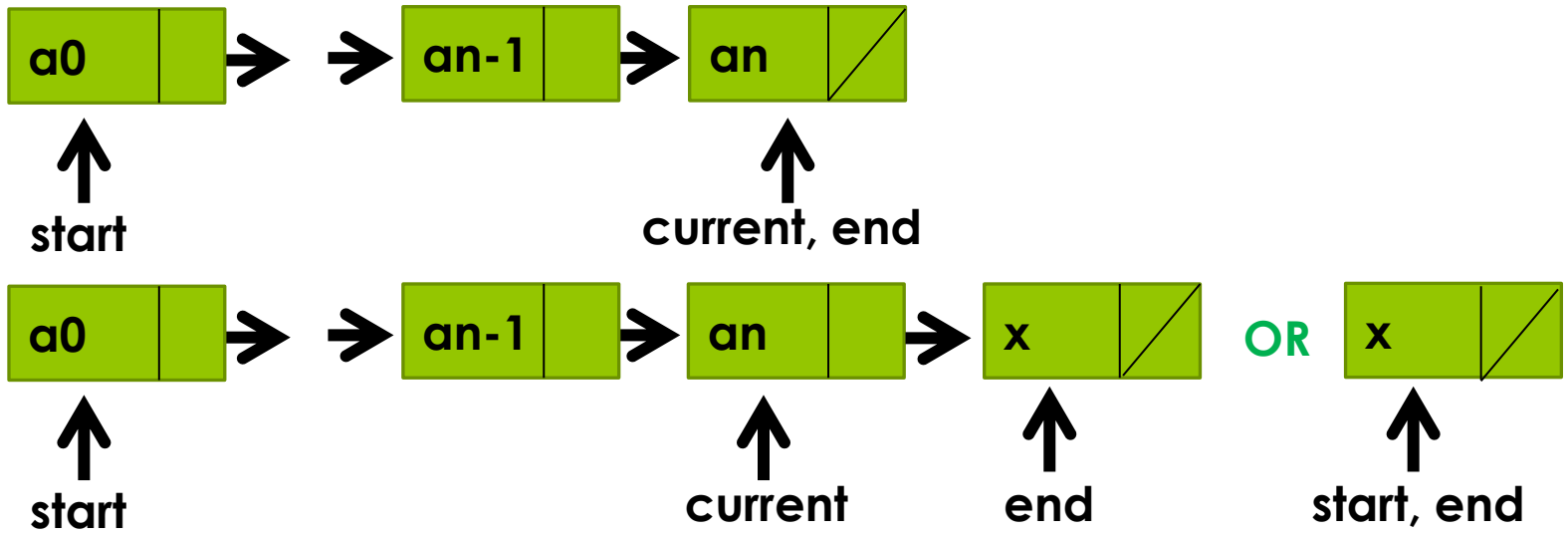
```
///
/// Copies all elements of a list
///
template <class T>
void LList<T>::CopyList(LList<T> const & list)
{
  start = end = NULL;
  if (list.start) {
    elem_link<T> *p = list.start;
    while(p) {
      ToEnd(p->inf);
      p = p->link;
    }
  }
}
```

Итератор

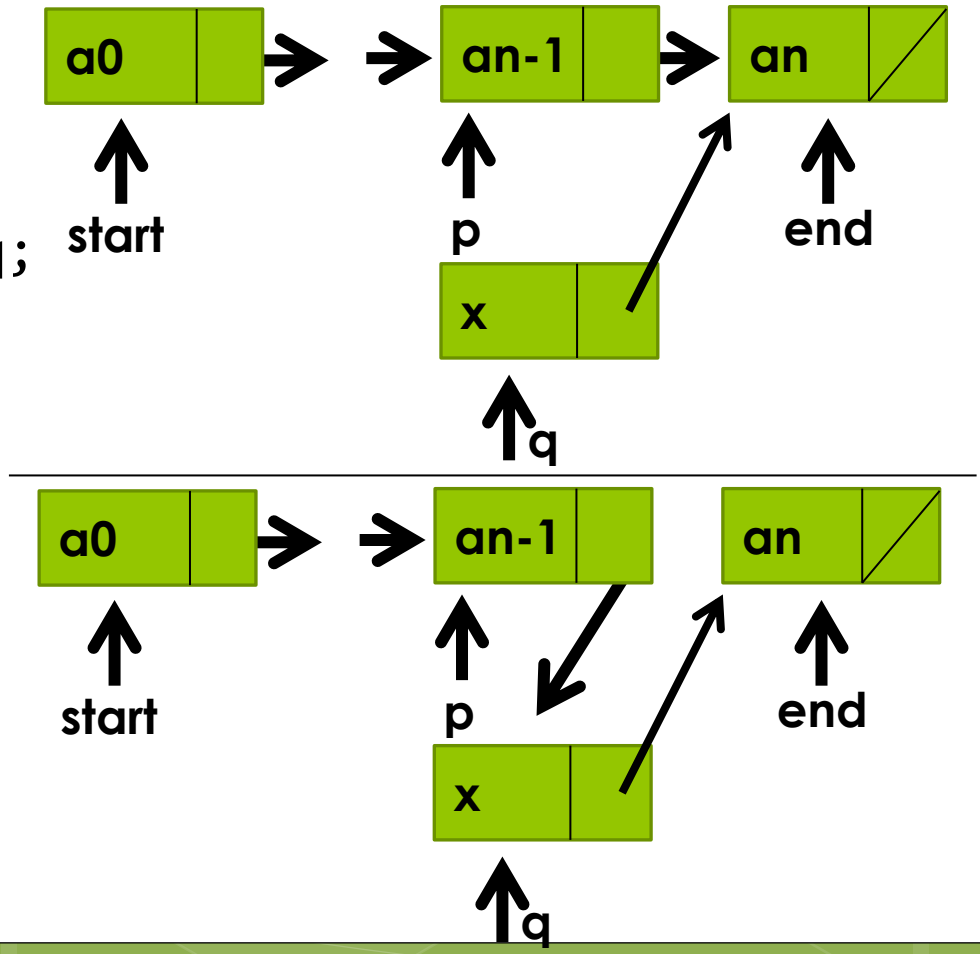
- Абстракция на указател към елемент на редица или по-точно на указател към елемент на контейнер (стек, опашка, свързан списък)

```
template <class T>
void LList<T>::IterStart(elem_link<T> *p)
{
  if (p) current = p;
  else current = start;
}


template <class T>
elem_link<T>* LList<T>::Iter()
{
  elem_link<T> *p = current;
  if (current) current = current->link;
  return p;
}
```

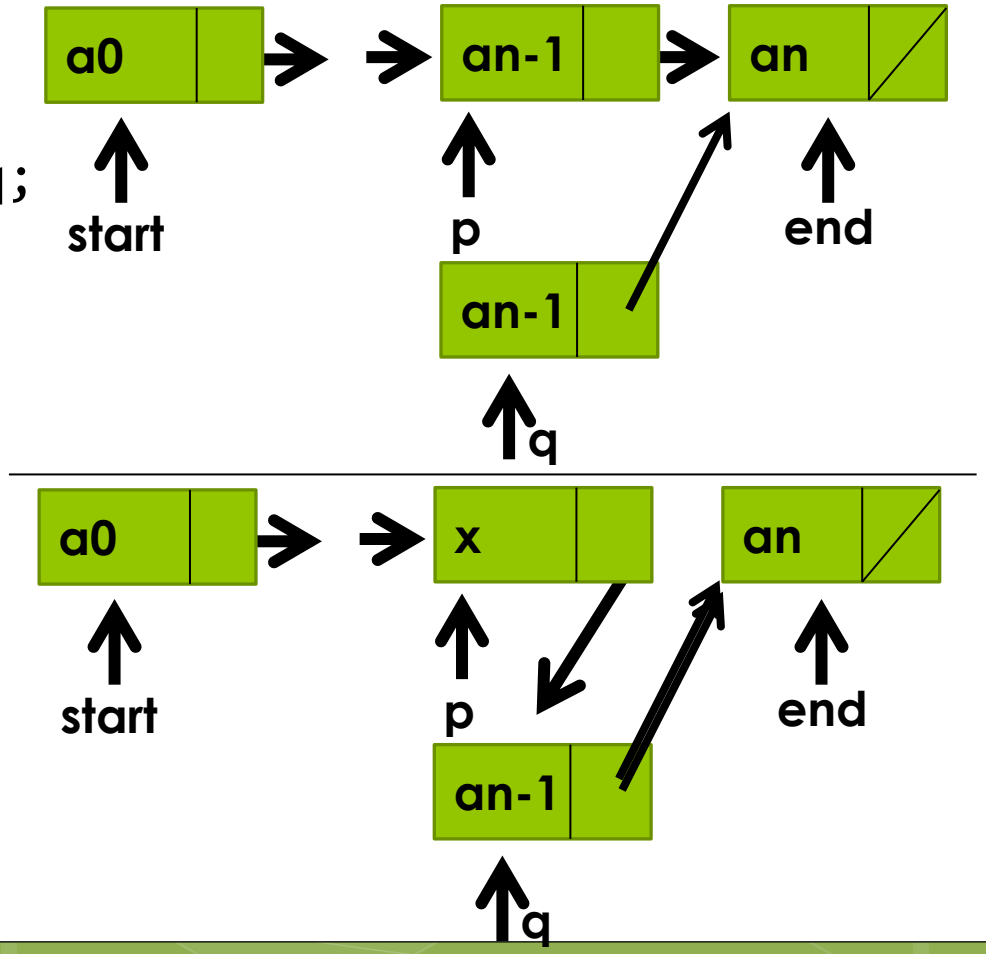
```
template <class T>
void LList<T>::ToEnd(T const & x)
{
  current = end;
  end = new elem_link<T>;
  end->inf = x;
  end->link = NULL;
  if (current) current->link = end;
  else start = end;
}
```



```
template <class T>
void LList<T>::InsertAfter(elem_link<T> *p, T const & x)
{
  elem_link<T> *q = new elem_link<T>;
  q->inf = x;
  q->link = p->link;

  if (p == end) end = q;

  p->link = q;
}
```



```
template <class T>
void LList<T>::InsertBefore(elem_link<T> *p, T const & x)
{
  elem_link<T> *q = new elem_link<T>;
  *q = *p;

  if (p == end) end = q;

  p->inf = x;
  p->link = q;
}
```



```
template <class T>
bool LList<T>::DeleteAfter(elem_link<T> *p, T & x)
{
  if (p->link) {
    elem_link<T> *q = p->link;
    x = q->inf;
    p->link = q->link;

    if (q == end) end = p;

    delete q;
    return 1;
  }

  return 0;
}
```

```
template <class T>
void LList<T>::DeleteElem(elem_link<T> *p, T & x)
{
  if (p == start) {
    x = p->inf;
    if (start == end) {
      start = end = NULL;
    }
    else start = start->link;

    delete p;
  }
  else {
    elem_link<T> *q = start;
    while(q->link != p) q = q->link;
    DeleteAfter(q, x);
  }
}
```

```
template <class T>
bool LList<T>::DeleteBefore(elem_link<T> *p, T & x)
{
  if (p != start) {
    elem_link<T> *q = start;
    while(q->link != p) q = q->link;
    DeleteElem(q, x);
    return 1;
  }

  return 0;
}
```

```
template <class T>
void LList<T>::print()
{
  elem_link<T>* p = start;
  while(p) {
    cout << p->inf << " ";
    p = p->link;
  }

  cout << endl;
}
```

```
template <class T>
int LList<T>::length()
{
  int n = 0;
  elem_link<T>* p = start;
  while(p) {
    n++;
    p = p->link;
  }

  return n;
}
```

```
// Реализация с итератор
template <class T>
int LList<T>::length()
{
  int n = 0;
  IterStart();
  elem_link<T> *p = Iter();
  while(p) {
    n++;
    p = Iter();
  }

  return n;
}
```

```
template <class T>
void LList<T>::concat(LList<T> const & list)
{
  elem_link<T> *p = list.start;
  while(p) {
    ToEnd(p->inf);
    p = p->link;
  }
}
```

```
// bad
template <class T>
void LList<T>::concat(LList<T> const & list)
{
  end->link = list.start;
  end = list.end;
}
```

```
LList<int> list;
list.ToEnd(1);
list.ToEnd(2);

//---------------------------------
elem_link<int> *p = list.Iter();
list.InsertBefore(p, 3);

list.IterStart();
p = list.Iter();

list.InsertAfter(p, 4);
list.print();

// 3 4 1 2
```

```
LList<int> list;
list.ToEnd(1);
list.ToEnd(2);

list.IterStart();
elem_link<int> *p = list.Iter();
list.InsertBefore(p, 3);


list.InsertAfter(p, 4);
list.print();

// 3 4 1 2
```

```
LList<int> list; int x;
list.ToEnd(1);
list.ToEnd(2);
list.IterStart();
elem_link<int> *p = list.Iter();
list.InsertBefore(p, 3);
list.InsertAfter(p, 4);
list.print();

list.DeleteElem(p, x);
list.print(); // 4 1 2

list.IterStart();
p = list.Iter();
list.DeleteAfter(p, x);
list.print(); // 4 2

list.IterStart();
p = list.Iter();
p = list.Iter();
list.DeleteBefore(p, x);
list.print(); // 2
```

```
LList<int> list; int x;
list.ToEnd(1);
list.ToEnd(2);
list.IterStart();
elem_link<int> *p = list.Iter();
list.InsertBefore(p, 3);
list.InsertAfter(p, 4);
list.print();

list.DeleteElem(p, x);
list.print(); // 4 1 2

list.DeleteAfter(p, x);
```

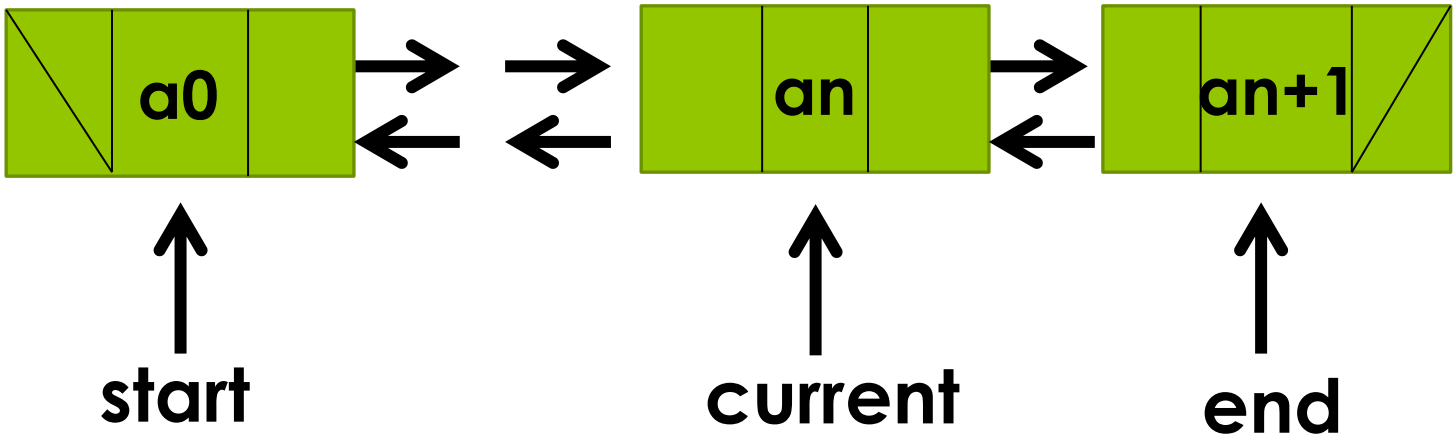
error

```
LList<int> list; int x;
list.ToEnd(1);
list.ToEnd(2);
list.IterStart();
elem_link<int> *p = list.Iter();
list.InsertBefore(p, 3);
list.InsertAfter(p, 4);
list.print();

list.DeleteElem(p, x);
list.print(); // 4 1 2

list.IterStart();
p = list.Iter();
list.DeleteAfter(p, x);
list.print(); // 4 2

p = list.Iter();
list.DeleteBefore(p, x);
list.print();
```

error

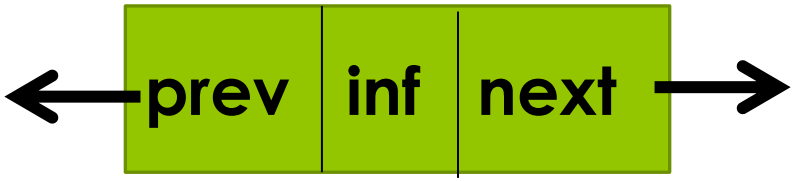
Свързан списък с две връзки

- Въвеждат се тройни кутии, с едно информационно и две свързващи полета, съдържащи текущия елемент и адресите на предшестващия и следващия го елементи
- Въвеждат се и указатели към края и към текущ елемент на списъка



Свързан списък с две връзки

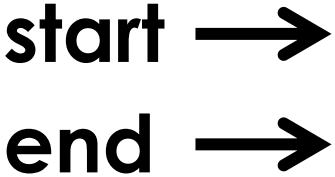
```
template <class T>
struct elem_link2 {
  T inf;
  elem_link2<T> *prev;
  elem_link2<T> *next;
};
```



```
template <class T>
class DList
{
private:
  elem_link2<T> *start;
  elem_link2<T> *end;
  elem_link2<T> *currentS;
  elem_link2<T> *currentE;

  void DeleteList();
  void CopyList(DList<T> const &);
```

```
public:
  DList();
  DList(Dlist<T> const &);
  DList& operator= (Dlist<T> const &);
  ~DList();

  void IterStart(elem_link2<T> *p = NULL);
  void IterEnd(elem_link2<T> *p = NULL);

  elem_link2<T>* IterNext();
  elem_link2<T>* IterPrev();

  void ToEnd(T const &);
  void DeleteElem(elem_link2<T>*, T &);

  void print();
  void print_reverse();
  int length();
};
```

```
///
/// Creates an empty list
///
template <class T>
DList<T>::DList()
{
  start = NULL;
  end = NULL;
}
```



```
///
/// Destroys a list
///
template <class T>
DList<T>::~DList()
{
  DeleteList();
}
```

```
///
/// Copy constructor
///
template <class T>
DList<T>::DList(Dlist<T> const & list)
{
  CopyList(list)
}
```

```
///
/// Copies the contents of one list to another
///
template <class T>
DList<T>& DList<T>::operator=(Dlist<T> const & list)
{
  if(this != &list)
  {
    DeleteList();
    CopyList(list);
  }
  return *this;
}
```

```
///
/// Removes all elements of a list
///
template <class T>
void DList<T>::DeleteList()
{
  elem_link2<T> *p;
  while(start) {
    p = start;
    start = start->next;
    delete p;
  }

  start = end = NULL;
}
```

```
///
/// Copies all elements of a list
///
template <class T>
void DList<T>::CopyList(DList<T> const & list)
{
  start = end = NULL;
  if (list.start) {
    elem_link2<T> *p = list.start;
    while(p) {
      ToEnd(p->inf);
      p = p->next;
    }
  }
}
```

```
template <class T>
void DList<T>::IterStart(elem_link2<T> *p)
{
  if (p) currentS = p;
  else currentS = start;
}


template <class T>
elem_link2<T>* DList<T>::IterNext()
{
  elem_link2<T> *p = currentS;
  if (currentS) currentS = currentS->next;
  return p;
}
```

```
template <class T>
void DList<T>::IterEnd(elem_link2<T> *p)
{
  if (p) currentE = p;
  else currentE = end;
}


template <class T>
elem_link2<T>* DList<T>::IterPrev()
{
  elem_link2<T> *p = currentE;
  if (currentE) currentE = currentE->prev;
  return p;
}
```

```
template <class T>
void DList<T>::ToEnd(T const & x)
{
  elem_link2<T> *p = end;
  end = new elem_link2<T>;
  end->inf = x;
  end->next = NULL;

  if (p) p->next = end;
  else start = end;

  end->prev = p;
}
```

```
template <class T>
void DList<T>::DeleteElem(elem_link2<T> *p, T & x)
{
  x = p->inf;
  if (start == end) {
    start = NULL;
    end = NULL;
  }
  else if (p == start) {
    start = start->next;
    start->prev = NULL;
  }
  else if (p == end) {
    end = p->prev;
    end->next = NULL;
  }
  else {
    p->prev->next = p->next;
    p->next->prev = p->prev;
  }
  delete p;
}
```

```
template <class T>
void DList<T>::print()
{
  elem_link2<T>* p = start;
  while(p) {
    cout << p->inf << " ";
    p = p->next;
  }

  cout << endl;
}
```

```
template <class T>
void DList<T>::print_reverse()
{
  elem_link2<T>* p = end;
  while(p) {
    cout << p->inf << " ";
    p = p->prev;
  }

  cout << endl;
}
```

```
template <class T>
int DList<T>::length()
{
  int n = 0;
  elem_link2<T>* p = start;
  while(p) {
    n++;
    p = p->next;
  }

  return n;
}
```

```
DList<int> list;
list.ToEnd(1);
list.ToEnd(2);
list.ToEnd(3);
list.ToEnd(4);

list.print();  // 1 2 3 4
list.print_reverse(); // 4 3 2 1
cout << list.length(); // 4
```

```
DList<int> list;
list.ToEnd(1);
list.ToEnd(2);
list.ToEnd(3);
list.ToEnd(4);

list.print(); // 1 2 3 4
list.print_reverse(); // 4 3 2 1
cout << list.length(); // 4

list.IterStart();
elem_link2<int> *p = list.IterNext();
list.DeleteElem(p, x);
list.print();  // 2 3 4

list.IterEnd();
elem_link2<int> *q = list.IterPrev();
list.DeleteElem(q, x);
list.print(); // 2 3
```

```
cout << “КРАЙ”;
```