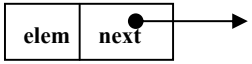
Linearne struktura - lista

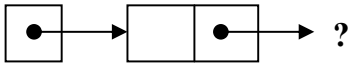
```
typedef elemType int;
typedef struct node Node;

typedef struct node
{
        elemType elem;    /* podatak iz kolekcije  */
        Node * next;      /* pokazuje na slijedeći node */
};
```

Ovakovu strukturu vizuelno možemo predstaviti slijedećom slikom



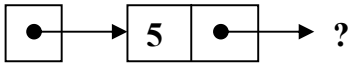
Proces stvaranja jednostavne liste započet ćemo alociranjem memorije za strukturu node.

```
Node *First = new Node;
```

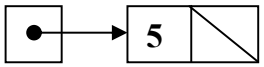


Sada možemo upisati neki podatak, primjerice

```
First -> elem = 5;
```



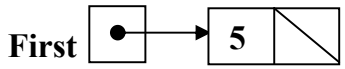

```
First -> next = NULL;
```



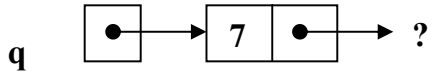
Kako dodati još jedan element ?

Kako dodati još jedan element ?
Prvo, formiramo element tipa Node, kojem pristupamo pomoću pokazivača q.

```
Node * q =  new Node;
```

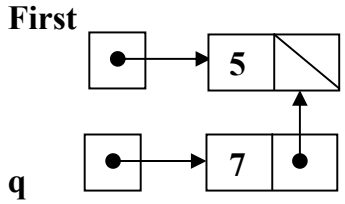



```
q ->elem =7;
```

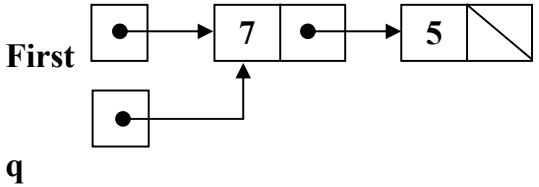


Zatim, ova dva elementa se "povežu" sljedećim naredbama:

```
q ->next = First;
```




```
First = q;
```



Novi element je postavljen na početak liste. Očito je da nam pokazivač q više nije potreban jer on ima sada istu vrijednost kao First pokazivač (q koristimo kao pomoćni pokazivač za formiranje vezane liste). Na ovaj način se može formirati lista s proizvoljnim brojem članova.

Vidimo da je početak liste (ili glava liste) zabilježen u pokazivaču kojeg se naziva First (tradicionalno se naziva i Head). Kraj liste se je obilježen s NULL vrijednošću "next" pokazivača posljednjeg elementa liste.

Funkcija: prepend

```
/* funkcija: prepend dodaje element na glavu liste
 * Argumenti:    first – pokazivač na glavu liste
 *               pNew -  pokazivač na element kojeg dodajemo
 * Vraća: pokazivač na glavu liste
 */

Node * prepend(Node * first, Node * pNew)
{
   if(pNew != NULL) /* izvrši samo ako je alociran element */
      pNew->next = first;
   return(pNew);    /* new postaje first pokazivač*/
}
```

Funkciju prepend koristimo prema slijedećem obrascu:

1. alociramo memoriju za element liste **Node * pNew = pNew Node;**

2. postavljamo sadržaj elementa **pNew->elem = ..**

3. dodajemo element u listu **First = prepend(First, pNew);**

4. ako je `First != NULL` operacija je uspjela.

Šetnja po listi

Ako znamo pokazivač glave liste uvijek možemo odrediti pokazivač na sljedeći element pomoću "next" pokazivača.

```
Node * Ptr = First->next;
```

Dalje, sukscesivno možemo usmjeravati pokazivač `Ptr` na slijedeći element liste naredbom

```
Ptr = Ptr->next;
```

Na taj se način može pristupiti svim elementima liste. Taj postupak ćemo zvati šetnja po listi (list traversing). Šetnja po listi završava kada je `Ptr == NULL`.

Često je razlog za "šetnju po listi" traženje po elementima liste. Koristimo funkciju:

```
Node * find_in_list(elemType x, Node * first )
{
      //ispituje da li lista sadrži elem x
      Node * p = first;
      while( p != NULL && p->elem != x )
          p = p->next;
      return p;
}


/*ispis elemenata liste od glave prema repu*/

void print_list(Node * l )
{
     while( l != NULL) {
          cout << l->elem;
          l = l->next;
     }
}
```

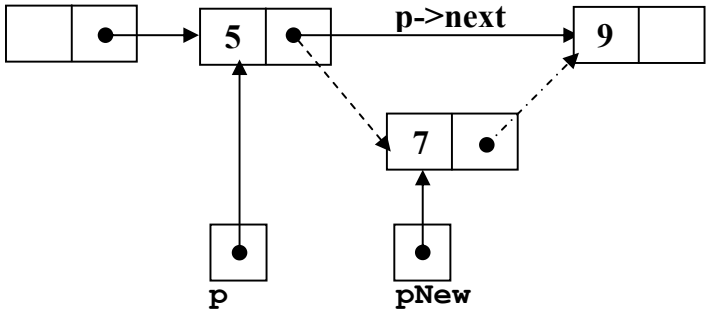
14.1.2 Dodavanje elementa na kraj liste

Ako lista još nije formirana, tj. ako je `First==NULL`, koristi se postupak opisan u funkciji prepend. U suprotnom, odredi se pokazivač krajnjeg elementa liste . Taj pokazivač, nazovimo ga last, ima karakteristiku da je `last->next == NULL`. Zatim se `last->next` postavlja na vrijednost pokazivača elementa kojeg dodajemo u listu. Pošto dodani element predstavlja rep liste njega zaključujemo s `NULL` pokazivačem.

```
Node * append(Node * first, Node * pNew)
{
   Node * last = first;              /* start sa last = first */
   if(first == NULL) {               /* ako lista nije formirana */
      first = pNew;                  /* iniciramo first pokazivač */
      pNew -> next = NULL;
   }
   else {
      while ( last->next != NULL)    /* 1. odredi last pokazivač */
          last = last -> next;
      last -> next = pNew;           /* 2. dodaj "pNew" element   */
      pNew -> next = NULL;           /* 3. označi kraj liste     */
   }
   return first;                     /* pokazivač na glavu liste*/
 }
```

Umetanje elementa u listu

Pretpostavimo da postoji kreirana lista, i da želimo umetnuti element iza elementa na kojeg pokazuje "p". Pokazivač pNew neka pokazuje na element kojeg želimo umetnuti. Ilustrirajmo to slikom:



Umetanje provodimo tako da najprije elemente "7" i "9" vežemo naredbom:

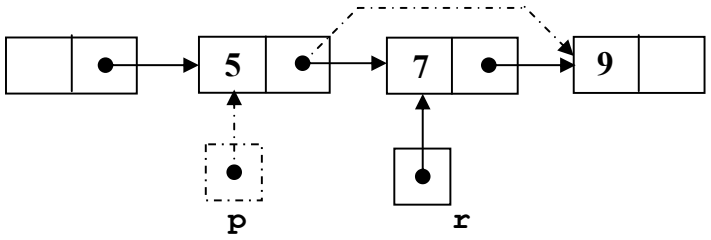
```
pNew -> next = p -> next;
```

a zatim elemente "5" i "7" vežemo naredbom: `p -> next = pNew;`
Ako bi postupili obrnutim redosljedom veza elemenata bi bila prekinuta.

```
Node * node_insert_after(Node * this, Node * pNew)
{
     if (this == NULL)
          pNew->next = NULL;
     else {
          pNew->next = this->next;
          this->next = pNew;
     }
     return(pNew);
}
```

14.1.4 Odstranjivanje (brisanje) elementa liste:

Razmotrimo slijedeći slučaj: Želimo odstraniti element na kojeg pokazuje "r".



Uočimo da je pokazivač `r` jednak `p->next`. Ako znamo pokazivač na prethodni element odstranjivanje elementa se provodi tako da njega vežemo na element koji slijedi iza elementa kojeg odstranjujemo, tj.

```
p->next = r->next;
```

Također, moramo dealocirati memoriju koju je taj element zauzimao s delete r;

<u>Uočite: element možemo odstraniti samo ako odredimo pokazivač na element prije njega.</u>

```
Node * find_previous_node(Node * first, Node * this )
{
    Node *     previous, current;
    current = previous = first ;   /* start at first */
    while( (current != this) && (current->next != NULL ))   {
        previous = current;
        current = current->next ;
    }
    return (previous) ;          /*  return address or NULL on error */
}
```

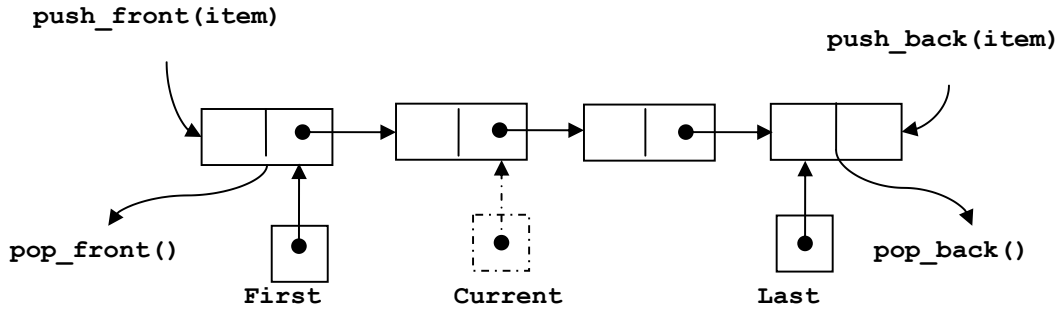
```
int remove_node(Node **pfirst, Node * This)
{
    if(*pfirst == NULL || this == NULL) return FALSE;
    if(*pfirst == This && pfirst !=NULL) /* remove first  if not NULL*/
          *pfirst = this->next;
    else  {
       Node * prev = find_previous_node(*pfirst, this );
       if(prev == NULL) return FALSE;
       prev -> next = this->next;
    }
    delete this;
    return TRUE;
}
```

Upamtimo:

- **Lista je dinamička strukutura. Povećava se i smanjuje prema zahtjevima korisnika.**
- **Pokazano je da se operacije dodavanja i brisanja elemenata liste provode jednostavnije nego je to slučaj s nizovima.**
- **Umetanje elementa unutar liste je relativno spora operacija jer se u njoj mora odrediti i položaj elementa koji prethodi mjestu umetanja. Znatno je brže umetanje elemenata na glavi liste.**
- **Liste treba smatrati kolekcijama s "sekvencijalnim pristupom", za razliku od nizova koje koristimo kao kolekcije sa "slučajnim pristupom".**

14.2. ADT List

U prethodnoj implementaciji liste korišten je pristup uobičajen za C jezik. Loša strana ovog rješenja je da korisnik mora znati kako je lista izgrađena. Sada ćemo definirati ADT i generičku klasu `List` koja će nam služiti kao opći kontenjer u kojeg se može ubacivati elemente sprijeda i straga.



Specifikacija ADT List

- **push_front(el)** - **element el se ubacuje na početak liste**
- **pop_front()** - **briše se element koji se nalazi na početku liste**
- **push_back(el)** - **element el se dodaje na kraj liste**
- **pop_back()** - **briše se element koji se nalazi na kraju liste**
- **remove(el)** - **odstranjuje elemente iz liste koji su jednaki elementu el.**
- **is_present(el)** - **vraća true ako je element el u listi**
- **clear()** - **brišu se svi elementi liste**
- **size()** - **vraća broj elemenata u listi**
- **empty()** - **vraća true ako je lista prazna**

Funkcije za "šetnju" po listi, tj. za dobavu elemenata liste, su

- **to_front()** - **postavlja na početak liste**
- **get_next()** - **vraća pokazivač na slijedeći element**
- **first_elem()** - **vraća pokazivač na prvi element liste**
- **last_elem()** - **vraća pokazivač na posljednji element liste**

14.2.2 Specifikacija i implementacija generičke klase List

```
template <class Type> class List
{
public:
  List()  { First = Last = Current = NULL; }
  ~List() { clear(); }

  void    push_front(const Type & elem);  // elem se ubacuje na početak
  void    pop_front();              // briše se  elem s početka liste
  void    push_back(const Type & elem);   // elem se ubacuje na kraj
  void    pop_back();               // briše se elem s kraja liste
  int     remove(Type &elem);       // odstranjuje element elem
  bool    is_present(Type &elem) ;  // vraća true ako je elem u listi
  void    clear();                  // brišu se svi elementi liste
  int     size();                   // vraća broj elemenata u listi
  bool    empty()     { return First == NULL ? true : false; }

  // funkcije za "šetnju" po listi
  void    to_front()    { Current = NULL; }         // postavlja na početak liste
  Type   *get_next();   // vraća pokazivač na slijedeći elem.
  Type   *first_elem()  { return (First == NULL) ? NULL : &(First->Elem);}
  Type  *last_elem()    { return (Last == NULL) ? NULL : &(Last->Elem);}

protected:
  class ListNode
  {
  public:
    ListNode(const Type &elem, ListNode *list = NULL) : Elem(elem) { Next = list; }
    Type      Elem;
    ListNode *Next;
  };

  ListNode *Current;
  ListNode *First;
  ListNode *Last;
};
```

14.2.3 Realizacija članskih funkcija klase List

```
template <class Type>  void List<Type>::push_front(const Type &elem)
{
  ListNode *newNode = new ListNode(elem,First);
  assert(newNode != NULL);
  if (First == NULL) Last = newNode;
  First = newNode;
}


template <class Type>  void List<Type>::pop_front()
{
     if(empty())
          return;
     ListNode *tmp = First;
     First = First->Next;
     delete tmp;
}

template <class Type>  void List<Type>::pop_back()
{
     if(empty())  return;
     if(First == Last)
          pop_front();
     else {
         ListNode *p = First;
          while (p->Next != Last)
          {
               p = p ->Next;
          }
          delete Last;
          Last = p;
          Last->Next=NULL;
     }
}
```

```
template <class Type> void List<Type>::push_back(const Type &elem)
{
  ListNode *newNode = new ListNode(elem);
  assert(newNode != NULL);

  if (First == NULL) First = newNode;
  else    Last->Next = newNode;
  Last = newNode;
}

template <class Type> void List<Type>::clear()
{
  ListNode *pNode = First;

    while (pNode != NULL){
        ListNode *tmp = pNode;
        pNode = pNode->Next;
        delete tmp;
    }
    First = Last = NULL;
}

template <class Type>  int List<Type>::size()
{
  int count = 0;
  Current = First;
  while (Current != NULL)  {
      count++;
      Current = Current->Next;
  }

  Current = NULL;
  return count;
}
```

```
template <class Type> int List<Type>::remove_elem(Type &elem)
{
  ListNode *pNode = First;
  int count = 0;

  while ( pNode != NULL && pNode->Elem == elem) {
      ListNode *tmp = pNode->Next;
      delete pNode;
      count++;
      pNode = tmp;
    }

  if ((First = pNode) == NULL) {
      Last = NULL;
      return count;
    }

  ListNode *prevNode = pNode;
  pNode = pNode->Next;

  while (pNode != NULL) {
      if ( pNode->Elem == elem) {
         prevNode->Next = pNode->Next;
         if (Last == pNode) Last = prevNode;
         delete pNode;
         count++;
         pNode = prevNode->Next;
      }
      else {
         prevNode = pNode;
         pNode = pNode->Next;
     }
    }
  return count;
}
```

```
template <class Type> Type * List<Type>::get_next()
{
  if (Current == NULL)
    Current = First;
  else
    Current = Current->Next;

  if (Current != NULL)
    return &(Current->Elem);
  else
    return NULL;
}


template <class Type>  bool List<Type>::is_present(Type &elem)
{
  if (First == NULL) return false;
  ListNode *pNode = First;
  for (;pNode != NULL; pNode = pNode->Next)
    if (pNode->Elem == elem) return true;
  return false;
}
```

14.2.4 Testiranje klase List

U programu test-list.cpp testirat ćemo operacije:

1 - za unos elemenata na početak liste
2 - za unos elemenata na kraj liste,
3 - za brisanje početnog elementa,
4 - za brisanje krajnjeg elementa,
5 - za brisanje proizvoljnog elementa.

Komuniciranje s korisnikom je pomoću prikladnog izbornika.

```
#include <iostream.h>
#include "list.h"

// koristit ćemo listu cijelih brojeva
typedef List<int> IntList;

// ispis liste
void print_list(IntList &L)
{
   int *p;
   L.to_front();
   while(p = L.get_next())  cout << *p << ",";
   cout << endl;
}

// Izbornik
void instructions(void)
{
   cout << "Odaberi :\n"
           "   1 - za unos elemenata na pocetak liste.\n"
           "   2 - za unos elemenata na kraj liste..\n"
           "   3 - za brisanje pocetnog elementa.\n"
           "   4 - za brisanje krajnjeg elementa.\n"
           "   5 - za brisanje proizvoljnog elementa.\n"
           "   0 - za kraj.\n";
}
```

```
int main()
{
   IntList L;
   int choice, element;
   instructions(); cout << "? ";   cin >> choice;
   while (choice != 0)  {
      switch (choice) {
         case 1: // unos elementa na pocetak liste
            cout << "Upisi broj: "; cin >> element;
            L.push_front(element); print_list(L);
            break;
         case 2: // unos elementa na kraj liste
         cout << "Upisi broj: ";  cin >> element;
            L.push_back(element); print_list(L);
            break;
         case 3: // brisanje pocetnog elementa
            L.pop_front();  print_list(L);
            break;
         case 4: // brisanje krajnjeg elementa
            L.pop_back();   print_list(L);
            break;
         case 5: // brisanje proizvoljnog elementa
            cout << "Upisi broj koji zelis izbrisati: "; cin >> element;
            if(!L.is_present(element))
               cout << "Broj " << element <<" nije u listi!\n";
            else {
               L.remove(element); print_list(L);
            }
            break;
         default:
            cout << "Ponovi  izbor.\n"; instructions();
      }
      cout << "? ";  cin >> choice;
   }
   cout << "Kraj.\n";
   return 0;
}
```

STOG realiziran pomoću klase list

Sada ćemo pokazati da se može implementirati klasa Stack pomoću klase `List`.

```
// Klasa Stack realizirana pomoću liste
#include "List.h"

template <class T>  class Stack
{
   List<T> L;
public:
   Stack(){}
   ~Stack(){}

   bool empty()  {return L.empty();}
   // vraca true ako je stog prazan

   int size() {return L.size();}
   // vraca broj elemenata na stogu

   T& top() {return *L.first_elem();}
   const T& top() const {return *L.first_elem();}

   // PRE: na stogu mora biti bar jedan element
   // dobavlja vrijednost elementa na vrhu stoga u elem
   // ne odstranjuje element sa stoga

   void pop() { L.pop_front(); }
   // PRE: na stogu mora biti bar jedan elemant
   // odstranjuje top element sa stoga
   // POST: na stogu je jedan element manje

   void push(const T &elem){ L.push_front(elem); }
   // postavlja element na vrh stoga
   // POST: na stogu je jedan element vise

};
```

QUEUE – RED

Queue je struktura koja podsjeća na red za čekanje. Iz reda izlazi onaj koji je prvi u red ušao. Ovaj princip pristupa podacima se naziva FIFO – first in first out.
Temeljne operacije s ADT queue su:

```
#include "List.h"

template <class Type> class Queue
{
public:

   Queue(){}
   ~Queue(){}

   bool empty()  {return L.empty();}
   // vraca true ako je red prazan

   int size() {return L.size();}
   // vraca broj elemenata reda

    bool get(Type &elem);
   // PRE: u redu je bar jedan elemant
   // dobavlja vrijednost elementa koji je na početku reda
   // i odstranjuje ga iz reda
   // vraća true ako postoji element na stogu

   void put(const Type &elem){ L.push_back(elem); }
   // postavlja element na kraj reda
   // POST: u redu je jedan element vise

private:
   List<Type> L;
};
```

```
template <class Type> bool Queue<Type>::get(Type &elem)
{   if(L.empty())  return false;
   elem = *L.first_elem();
   L.pop_front();
   return true;
}
```

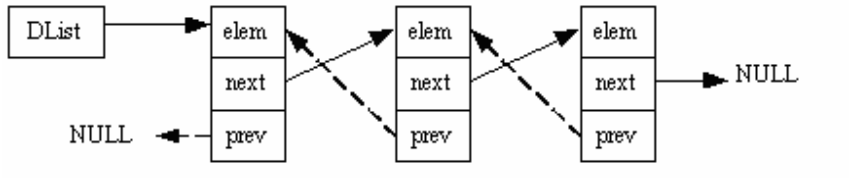
Testiranje klase Queue: Korisnik unosi proizvoljan broj stringova u kontenjer tipa reda - Queue. Unos završava kada se unese prazni string. Nakon toga se ispisuje sadržaj reda. Ispis će biti izvršen istim redom kako je izvršen i unos stringova.

```
#include <iostream>
#include <string>
#include "queue.h"
using namespace std;

typedef Queue<string> StringQueue;

void print_queue(StringQueue &Q)
{
  string s;
  while(Q.get(s)) cout << s << endl;
}

int main()
{
   StringQueue Q;
   string str;
   while (getline(cin, str))  {
      if(str.size()==0) break;
      Q.put(str);
   }
   cout << "\nOtkucali ste\n";
   print_queue(Q);
   return 0;
}
```

14.5 Dvostruko vezana lista

Ukoliko se čvor liste formira tako da sadrži dva pokazivača, `next` - koji pokazuje na sljedeći čvor i `prev` - koji pokazuje na prethodni čvor, dobije se dvostruko vezana lista. Realizira se pomoću sljedeće strukture podataka:

```
typedef int elemT;
typedef struct _node  Node;
typedef Node *DLIST;

struct _node
{
     elemT elem;   /* element liste  */
     Node *next;     /* pokazuje na sljedeći čvor */
     Node *prev;     /* pokazuje na prethodni čvor */
}
```



Slika 18.5 Dvostruko vezana lista

1. **Pokazivač `next` krajnjeg elementa i pokazivač `prev` glave liste su jednaki `NULL`.**
2. **Ovakvu listu se može iterativno obilaziti u oba smjera, od glave prema kraju liste (korištenjem pokazivača `next`) i od kraja prema početku liste (korištenjem pokazivača `prev`).**
3. **Umetanje unutar liste i brisanje iz liste se provodi jednostavnije i brže nego kod jednostruko vezane liste, jer je u svakom čvoru poznat pokazivač na prethodni čvor.**
4. **U odnosu na jednostruko vezanu listu, dvostruko vezana lista zauzima više memorije (zbog pokazivača `prev`) .**

Ako se vrši umetanje i brisanje čvorova samo na početku liste, tada je bolje rješenje koristiti jednostruko vezanu listu.

Kroz dva primjera bit će pokazano kako se implementira dvostruko vezana lista.

Primjer: **Prikana je implementacija funkcija za umetanje čvora na glavi liste `(dlist_add_front_node)` i odstranjivanje čvora na glavi liste `(dlist_delete_front_node)`. Uočite da je potrebno izvršiti nešto više operacija nego kod jednostruko vezane liste.**

```
void dlist_add_front_node(DLIST *pList, Node *n)
{
   if(n != NULL) /* izvrši samo ako je alociran čvor */
   {
      n->next = *pList;
      n->prev = NULL;
      if(*pList != NULL)
          (*pList)->prev = n;
      *pList = n;
   }
}

void dlist_delete_front_node(DLIST *pList)
{
   Node *tmp = *pList;
   if(*pList != NULL) {
       *pList = (*pList)->next;
      if(*pList != NULL)
          (*pList)->prev = NULL;
       freeNode(tmp);
   }
}
```

Zadatak: **Predthodnu klasu List realizirajte na način da sadrži dvostruko vezanu listu elemenata tipa T.**

14.6 Koncept standardne biblioteke - STL

U STL biblioteci, koja je opisana u Dodatku, primijenjen je sljedeći pristup:

1. **Podaci se nalaze u sekvencijalnim ili asocijativnim kontenjerima:**

- **Sekvencijalni kontenjeri su: string, vektor, list, deque (double ended queue)**
- **Asocijativni kontenjeri su: map, multimap, set i multiset**

2. **Obilazak kontenjera se postiže objektima koji se nazivaju iteratori. Iterator predstavlja poziciju u kontenjeru, a kod linearnih struktura ekvivalentan je pokazivaču na element kontenjera. Temelja logika primjene iteratora je:**

   Ako se u kontenjeru K nalaze objekti tipa T i ako varijabla x je tipa iteratora, tada

   x označava poziciju elementa iz kontenjera

   ***x** daje vrijednost elementa na poziciji x

   x++ daje poziciju sljedećeg elementa

   Početna i krajnjna pozicija iteratora se postavljaju funkcijama K.begin() i K.end().

Primjer: Za objekte tip vektor

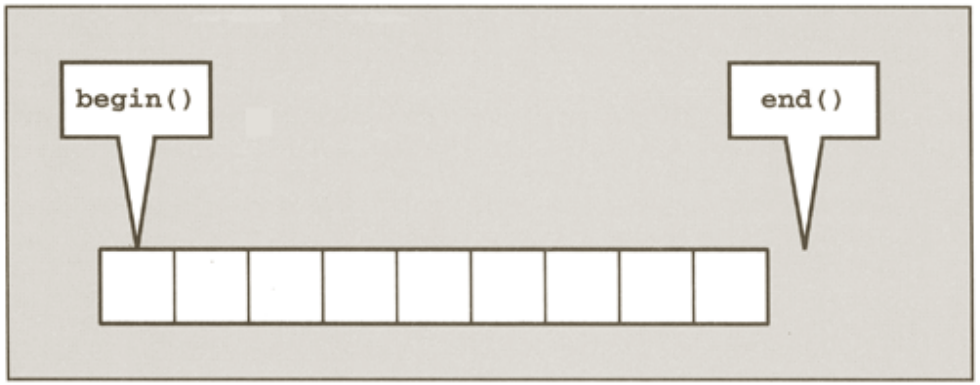
```
vector<int>    myVec;
```

Iterator se deklarira s:

```
vector<int>::iterator   pos;
```

Sada se iterator `pos` može koristiti za obilazak vektora:

```
// suma elemenata vektora
int sum = 0;
for ( pos = v.begin(); pos != v.end(); pos++ )
     sum += *pos;
```

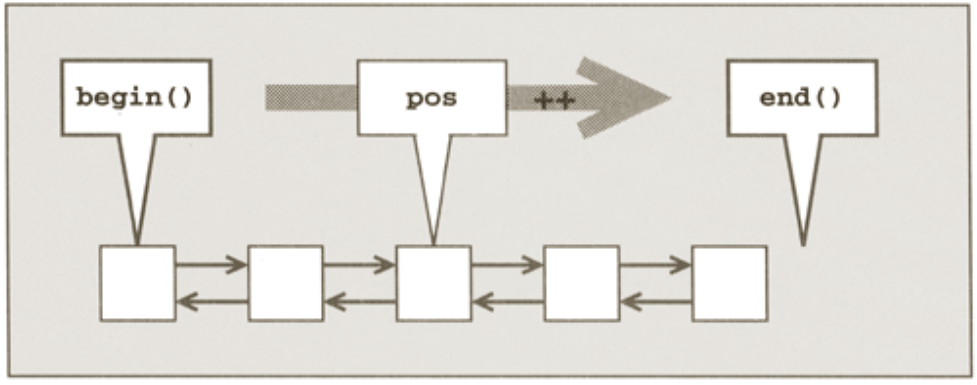


Istu logiku se može primijeniti za obilazak liste.

Primjer: U sljedećem programu se formira i ispisuje lista znakova

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
      list<char> L;  //list container for character elements

      // dodaj elemente od 'a' do 'z'
      for (char c='a'; c<='z'; ++c) {
            L.push_back(c);
      }
      // ispisi sve elemente

      list<char>::const_iterator pos;
      for (pos = L.begin(); pos != L.end(); ++pos) {
            cout << *pos << ' ' ;
      }
      cout << endl;
}
```

Primjetite da se ovdje koristi const_iterator objekt. To je tip iteratora kojim se pristupa elementima kolekcije, ali se ne mijenja njihova vrijednost.



Ako se želi mijenjati elemente liste tada se koristi prosti iterator, primjerice u prethodnoj se listi znakova svi znakovi pretvaraju u velika slova iteracijom:

```
//make all characters in the list uppercase
list<char>::iterator pos;
for (pos = L.begin(); pos != L.end(); ++pos) {
     *pos = toupper(*pos);
```

Kasnije će biti pokazana implementacija dvostruko vezane liste s iteratorima.

Tri su klase iteratora:

forward_iterator - omogućuje sekvencijalni obilazak u smjeru od početka do kraja kontenjera.
Dozvoljene operacije su: `a++, ++a, *a, a = b, a == b, a != b`

reverse_iterator - omogućuje sekvencijalni obilazak u smjeru od kraja do početka kontenjera
Početna i krajnja pozicija se dobiju funkcijema K.rbegin() i K.rend()

```
for (pos = v.rbegin();pos < v.rend(); pos ++ )
  t = *pos;
```

Dozvoljene operacije su: `a++, ++a, *a, a = b, a == b, a != b`

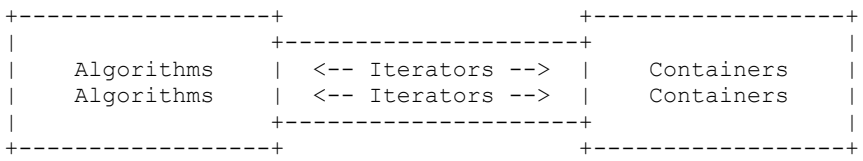
random_acess_iterator - omogućuje obilazak kontenjera u oba smjera i s proizvoljnim korakom, primjerice:

```
// suma parnih elemenata vektora
int sum_parni = 0;
for ( pos = v.begin(); pos < v.end(); pos = pos + 2 )
     sum_parni += *pos;
```

Dozvoljene operacije su: `a++, ++a, a--, --a, a += n, a -= n, a[n], a = b, a == b, a != b, a < b, a <= b, a > b, a >= b.`

Koncept iteratora je značajan jer omogućuje definiranje istih operacije za sve tipove kontenjera:

Operation	Effect
`Type c`	**Creates an empty container without any element**
`Type c1(c2)`	**Copies a container of the same type**
`Type c(beg,end)`	**Creates a container and initializes it with copies of all elements of [beg,end)**
`~ContType()`	**Deletes all elements and frees the memory**
`c.size()`	**Returns the actual number of elements**
`c.empty()`	**Returns whether the container is empty (equivalent to size()==0, but might be faster)**
`c.max_size()`	**Returns the maximum number of elements possible**
`c1 == 2`	**Returns whether c1 is equal to c2**
`c1 != c2`	**Returns whether c1 is not equal to c2 (equivalent to !(c1==c2))**
`c1 < c2`	**Returns whether c1 is less than c2**
`c1 > c2`	**Returns whether c1 is greater than c2 (equivalent to c2<c1**
`c1 <= c2`	**Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1))**
`c1 >= c2`	**Returns whether c1is greater than or equal to c2 (equivalent to !(c1<c2))**
`c1 = c2`	**Assigns all elements of c1 to c2**
`c1.swap(c2)`	**Swaps the data of c1and c2**
`swap(c1,c2)`	**Same (as global function)**
`c.begin()`	**Returns an iterator for the first element**
`c.end()`	**Returns an iterator for the position after the last element**
`c.rbegin()`	**Returns a reverse iterator for the first element of a reverse iteration**
`c.rend()`	**Returns a reverse iterator for the position after the last element of a reverse iteration**
`c.insert(pos,elem)`	**Inserts a copy of elem (return value and the meaning of pos differ)**
`c.erase(beg,end)`	**Removes all elements of the range [beg,end) (some containers return next element not removed)**
`c.clear()`	**Removes all elements (makes the container empty)**
`c.get_allocator()`	**Returns the memory model of the container**

3. **Važna je uloga iteratora u pristupu elementima kontenjera jer se oni u STL biblioteci koriste u različitim algoritmima**
4. 

<u>Algoritam</u> u STL biblioteci je "funkcija" kojom se uz pomoć iteratora djeluje na elemente kontenjera.

Te funkcije se mogu klasificirati u tri grupe: algoritmi nad sekvencama, algoritmi za sortiranje i numerički algoritmi

Algoritmi nad sekvencama

`count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition`

Algoritmi za sortiranje

`Sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexographical_compare, next_permutation, prev_permutation`

Numerički algoritmi

`Accumulate, inner_product, partial_sum, adjacent_difference`

Primjer: **u sljedećem programu prikazano je djelovanje algoritama:**

min_element() **- daje poziciju minimalnog elementa kolekcije**
max_element() **- daje poziciju maksimalnog elementa kolekcije**
sort() **- sortira lemente kolekcije**
find() **- daje poziciju elementa kolekcije**
reverse() **- uređuje kolekciju u obrnutom redoslijedu**
fill() **- postavlja sve elemente vektora na neku vrijednost**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
   vector<int> v;
   vector<int>::iterator pos;
   //formira vektor od 10 elemenata vrijednosti 0
   v.resize(10);
   fill(v.begin(), v.end(), 0);
   //insert elements from 1 to 6
   v.push_back(2); v.push_back(5);
   v.push_back(4); v.push_back(1);
   v.push_back(6); v.push_back(3);

   //find and print minimum and maximum elements
   pos = min_element (v.begin(), v.end());
   cout << "min: " << *pos << endl;
   pos = max_element (v.begin(), v.end());
   cout << "max: " << *pos << endl;

   //sort all elements
   sort (v.begin(), v.end());

   //find the first element with value 3
   pos = find (v.begin(), v.end(), 3);
   // reverse the order of the found element with value 3 and all
   // following elements
    reverse (pos, v.end());

  //print all elements
    for (pos=v.begin(); pos!=v.end(); ++pos) {
      cout << *pos << ' ' ;
    }
    cout << endl;
}
```

14.7 Kako su implementirani iteratori

Kod STL klase vektor iterator je drugo ime za pokazivač:

```
#include <algorithm>

template <class T> class vector {
public:
     typedef T * iterator;  //<<<<<<
     // constructors
     vector    () { buffer = 0; reserve(0); }
     vector    (unsigned int size) { buffer = 0; resize(size); }
     vector    (vector & v)
     { buffer = 0; resize(v.size()); copy(v.begin(), v.end(), begin());}
     ~vector   ()   { delete buffer; }
     // member functions
     T         back ()   { return buffer[mySize-1]; }
     iterator  begin     ()   { return buffer; }
     int       capacity  () { return myCapacity; }
     iterator  end       ()   { return buffer + mySize; }
     bool      empty     ()   { return mySize == 0; }
     T         front     ()   { return buffer[0]; }
     void      pop_back () { mySize--; }
     void      push_back (T value);
     void      reserve   (unsigned int newSize);
     void      resize    (unsigned int newSize){reserve(newSize); mySize = newSize;}
     int       size ()   { return mySize; }
     // operators
     T &       operator [ ] (unsigned int index) { return buffer[index]; }
private:
     unsigned int mySize;
     unsigned int myCapacity;
     T * buffer;
};
```

```
template <class T> void vector<T>::reserve (unsigned int newCapacity)
{
     if (buffer == 0) {
          mySize = 0;
          myCapacity = 0;
          }
     if (newCapacity <= myCapacity)
          return;
     T * newBuffer = new T [newCapacity];
     copy (buffer, buffer + mySize, newBuffer);
     myCapacity = newCapacity;
     delete buffer;
     buffer = newBuffer;
}

template <class T> void vector<T>::push_back (T value)
{
     if (mySize >= myCapacity)
          reserve(myCapacity + 5);
     buffer[mySize++] = value;
}
```

Implementacija iteratora u klasi list

Kod implementacije iteratora u drugim klasama treba voditi računa da se osigura funcionalnost pokazivača. Pokazat ćemo primjer definiranja STL klase list **(koja je realizirana kao dvostruko vezana lista), u pojednostavljenom obliku.**

Definirana je klasa `listIterator` koja se u klasi `list` koristi kao `iterator` (typedef). Ta klasa ima funkcionalnost pokazivača, jer je definiran operator indirekcije *, također su definirani operatori inkrementiranja pokazivača, jednakosti i pridjele vrijednosti.
Čvor liste je definiran klasom `node`, a lista je definirana klasom `list`.

```
template <class T> class listIterator
{
     typedef listIterator<T> iterator;
public:
     // constructor
     listIterator (list<T> * tl, node<T> * cl):theList(tl), currentNode(cl) {}

     T & operator * () { return currentNode->value; }
     void operator = (iterator & rhs)
          { theList = rhs.theList; currentNode = rhs.currentNode; }
     bool operator == (const iterator rhs) const
          { return currentNode == rhs.currentNode; }
     iterator & operator ++ (int)
          { currentNode = currentNode->nextNode; return * this; }
     iterator operator ++ ();
     iterator & operator -- (int)
          { currentNode = currentNode->prevNode; return * this; }
     iterator operator -- ();

protected:
     list <T> * theList;
     node <T> * currentNode;
     friend class list<T>;
};
```

```
template <class T>  class list {
public:
     typedef T value_type;
     typedef listIterator<T> iterator;

     // constructor and destructor
     list () : firstNode(0), lastNode(0) { }
     list (list<T> & x) : firstNode(0), lastNode(0) { }
     ~ list ();
     // operations
     bool empty () { return firstNode == 0; }
     int size();
     T & back () { return lastNode->value; }
     T & front () { return firstNode->value; }
     void push_front(T &);
     void push_back(T &);
     void pop_front ();
     void pop_back ();
     iterator begin () { return iterator (this, firstNode); }
     iterator end () { return iterator (this, 0); }
     void insert (iterator &, T &);
     void erase (iterator & itr) { erase (itr, itr); }
     void erase (iterator &, iterator &);
protected:
     node <T> * firstNode;
     node <T> * lastNode;
};

template <class T> class node {
     node (T & v) : value(v), nextNode(0), prevNode(0) { }
     T value;
     node<T> * nextNode;
     node<T> * prevNode;
     // allow lists to see element values
     friend class list<T>;
     friend class listIterator<T>;
};
```

```
template <class T> int list<T>::size ()
     // count number of elements in collection
{
     int counter = 0;
     for (node<T> * ptr = firstNode; ptr != 0; ptr = ptr->nextNode)
          counter++;
     return counter;
}


template <class T> void list<T>::push_front (T & newValue)
     // add a new value to the front of a list
{
     node<T> * newNode = new node<T> (newValue);
     if (empty())
          firstNode = lastNode = newNode;
     else {
          firstNode->prevNode = newNode;
          newNode->nextNode = firstNode;
          firstNode = newNode;
          }
}

template <class T> void list<T>::pop_front()
     // remove first element from list
{
     node <T> * save = firstNode;
     firstNode = firstNode->nextNode;
     if (firstNode != 0)
          firstNode->prevNode = 0;
     else
          lastNode = 0;
     delete save;
}
```

```
template <class T> list<T>::~list ()
     // remove each element from the list
{
     node <T> * first = firstnode;
     while (first != 0) {
          node <T> * next  = first->nextNode;
          delete first;
          first = next;
          }
}

template <class T> listIterator<T> listIterator<T>::operator ++ ()
     // postfix form of increment
{
          // clone, then increment, return clone
     listIterator<T> clone (theList, currentNode);
     currentNode = currentNode->nextNode;
     return clone;
}

template <class T> void list<T>::insert (listIterator<T> & itr, T & value)
     // insert a new element into the middle of a nodeed list
{
     node<T> * newNode = new node(value);
     node<T> * current = itr->currentNode;

     newNode->nextNode = current;
     newNode->prevNode = current->prevNode;
     current->prevNode = newNode;
     current = newNode->prevNode;
     if (current != 0)
          current->nextNode = newNode;
}
```

```
template <class T>
void list<T>::erase (listIterator<T> & start, listIterator<T> & stop)
// remove values from the range of elements
{
     node<T> * first = start.currentNode;
     node<T> * prev = first->prevNode;
     node<T> * last = stop.currentNode;
     if (prev == 0) {    // removing initial portion of list
          firstNode = last;
          if (last == 0)
               lastNode = 0;
          else
               last->prevNode = 0;
          }
     else {
          prev->nextNode = last;
          if (last == 0)
               lastNode = prev;
          else
               last->prevNode = prev;

          }
          // now delete the values
     while (start != stop) {
          listIterator<T> next = start;
          ++next;
          delete start.currentNode;
          start = next;
          }
}
```