Πίνακες – Λίστες

Γιάννης Θεοδωρίδης, Νίκος Πελέκης, Άγγελος Πικράκης
Τμήμα Πληροφορικής

Πίνακες (arrays)

- Ακολουθία μεταβλητών
  - με το ίδιο όνομα,
  - με τον ίδιο τύπο δεδομένων,
  - σε συνεχόμενες θέσεις μνήμης.
- Παράδειγμα

```
int a[50];
double d1[5], d2[10];
```

- Η αρίθμηση αρχίζει από το 0 !

```
for (i = 0; i < 50; i++)
   a[i] = i;
```

Πίνακες (arrays)

- Αρχικοποίηση πινάκων

```
int a[3] = {6, 7, 42};
char c[] = {'a', 'b', 'c', 'd', 'e'};
```

- Συμβολοσειρές
  - Είναι πίνακες χαρακτήρων

```
char s[6] = "abcde";
```

  - Τερματίζονται με τον κενό χαρακτήρα `'\0'`

```
char s[6] = {'a', 'b', 'c',
             'd', 'e', '\0'};
```


Πίνακες (arrays)

- Παράδειγμα

```
int a[3] = {5, 6, 7};
int b[3] = {2, 3, 1};
int i;

for (i = 0; i < 3; i++)
   printf("%d times %d is %d\n",
      a[i], b[i], a[i]*b[i]);
```

```
5 times 2 is 10
6 times 3 is 18
7 times 1 is 7
```

Πολυδιάστατοι πίνακες

- Παράδειγμα

```
int a[3][5];
char c[5][10][4];
```

- Αρχικοποίηση

```
int i[3][3] = { {1, 0, 0},
                {0, 1, 0},
                {0, 0, 1} };
char s[][10] = {
   "my", "name", "is", "joe"
};
```


Πολυδιάστατοι πίνακες

- Παράδειγμα: πολλαπλασιασμός πινάκων

```
double a[3][4] = ... ,
       b[4][5] = ... ,
       c[3][5];
int i, j, k;

for (i=0; i<3; i++)
   for (j=0; j<5; j++) {
      c[i][j] = 0.0;
      for (k=0; k<4; k++)
         c[i][j] += a[i][k] * b[k][j];
   }
```

Πίνακες ως ΑΤΔ

- Βασική πράξη: προσπέλαση στοιχείου

  a[i]
- Συνήθως υλοποιούνται με κάποιο ΣΤΔ πινάκων (arrays)
  - Κόστος προσπέλασης: O(1)
- Ο ΣΤΔ του μονοδιάστατου πίνακα επαρκεί για την υλοποίηση κάθε ΑΤΔ πίνακα
  - Συνάρτηση *loc* υπολογίζει τη θέση ενός στοιχείου του ΑΤΔ πίνακα στο μονοδιάστατο ΣΤΔ πίνακα της υλοποίησης

Πίνακες ως ΑΤΔ

- ΑΤΔ πίνακα δύο διαστάσεων $n \times m$

$i = 1$	0	1	2	3	4	5
$i = 2$	6	7	8	9	10	11
$i = 3$	12	13	14	15	16	17
	$j = 1$	2	3	4	5	6

$n = 3$

$m = 6$

loc $(n, m, i, j) = m\,(i - 1) + j - 1$

- Αρίθμηση κατά στήλες

  loc $(n, m, i, j) = n\,(j - 1) + i - 1$

Πίνακες ως ΑΤΔ

- ΑΤΔ κάτω τριγωνικού πίνακα $n \times n$

$i = 1$	0				
$i = 2$	1	2			
$i = 3$	3	4	5		
$i = 4$	6	7	8	9	
$i = 5$	10	11	12	13	14
	$j = 1$	2	3	4	5

$n = 5$

$loc\ (n, i, j) = i\ (i - 1) / 2 + j - 1$

- Ομοίως για συμμετρικούς πίνακες

Πίνακες ως ΑΤΔ

- ΑΤΔ τριδιαγώνιου πίνακα $n \times n$

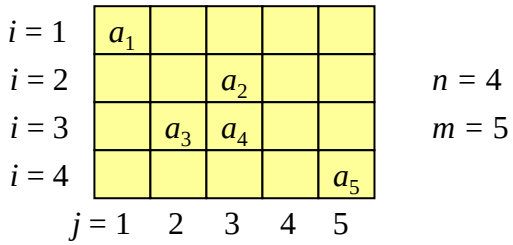
$i = 1$	0	1			
$i = 2$	2	3	4		
$i = 3$		5	6	7	
$i = 4$			8	9	10
$i = 5$				11	12
	$j = 1$	2	3	4	5

$n = 5$

$loc\ (n, i, j) = 2\ i + j - 3$

Πίνακες ως ΑΤΔ

- ΑΤΔ αραιού πίνακα $n \times m$



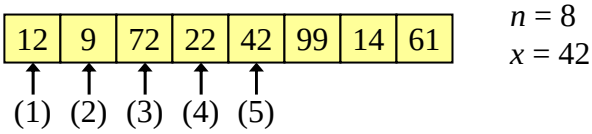
  - Υλοποίηση με δυαδικό πίνακα
  - Υλοποίηση με τρεις πίνακες

    row = [1, 2, 3, 3, 4] *col* = [1, 3, 2, 3, 5]

    val = [a_1, a_2, a_3, a_4, a_5]

Αναζήτηση σε πίνακες

- Σειριακή αναζήτηση
  - Τα στοιχεία διατρέχονται κατά σειρά
  - Κόστος: O(*n*)



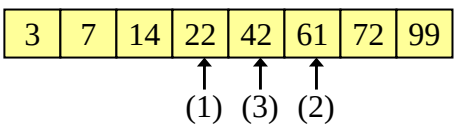
Αναζήτηση σε πίνακες

- Υλοποίηση σε C

```
int ssearch (int a[], int n, int x)
{
   int i;

   for (i = 0; i < n; i++)
      if (a[i] == x)
         return i;
   return -1;
}
```


Αναζήτηση σε πίνακες

- Δυαδική αναζήτηση
  - Ο πίνακας πρέπει να είναι ταξινομημένος
  - Κόστος: O(log*n*)



$n = 8$
$x = 42$

- Άλλες μέθοδοι αναζήτησης
  - Μικρότερο κόστος ⇒ περισσότερος χώρος
  - Πίνακες κατακερματισμού (hash tables)

Γραμμικές Λίστες (linear lists)

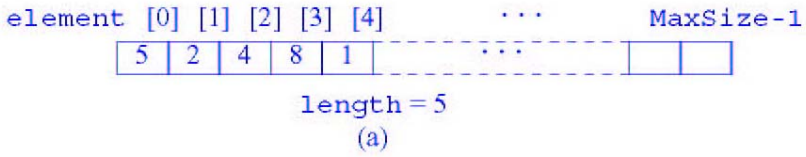
- Μια γραμμική λίστα είναι μια πεπερασμένη, ταξινομημένη ακολουθία δεδομένων
- Βασική αρχή: Τα στοιχεία της λίστας έχουν μια θέση μέσα στην ακολουθία (1ο, 2ο, 3ο κοκ.)
- Συμβολισμός: $<e_1, e_2, \ldots, e_n>$
- Ποιες λειτουργίες πρέπει να υλοποιήσουμε:
  - Δημιουργία μιας γραμμικής λίστας
  - Καθορισμός του αν η λίστα είναι άδεια, καθορισμός του μήκους της λίστας
  - Εύρεση του k-οστού στοιχείου, διαγραφή του k-οστού στοιχείου, εισαγωγή νέου στοιχείου ακριβώς μετά το k-οστό
  - Αναζήτηση ενός στοιχείου x μέσα στη λίστα

ΑΤΔ Γραμμική λίστα

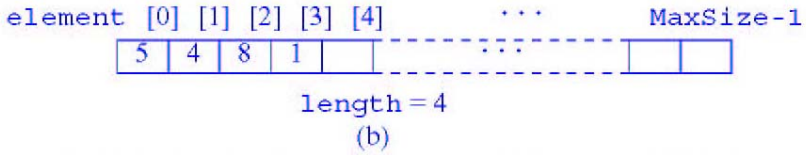
AbstractDataType *LinearList* {
  instances
    ordered finite collections of zero or more elements
  operations
    Create (): create an empty linear list
    Destroy (): erase the list
    IsEmpty (): return `true` if the list is empty, `false` otherwise
    Length (): return the list size (i.e., number of elements in the list)
    Find (*k*, *x*): return the *k*th element of the list in *x*
                return `false` if there is no *k*th element
    Search (*x*): return the position of *x* in the list
                return `0` if *x* is not in the list
    Delete (*k*, *x*): delete the *k*th element and return it in *x*
                function returns the modified linear list
    Insert (*k*, *x*): insert *x* just after the *k*th element
                function returns the modified linear list
    Output (*out*): put the list into the output stream *out*;
}

Εναλλακτικοί τρόποι αναπαράστασης

- Αναπαράσταση με βάση τον τύπο (ή στατική)
  - Χρησιμοποιούμε πίνακα για να αναπαραστήσουμε τα στοιχεία της λίστας. Παράδειγμα:



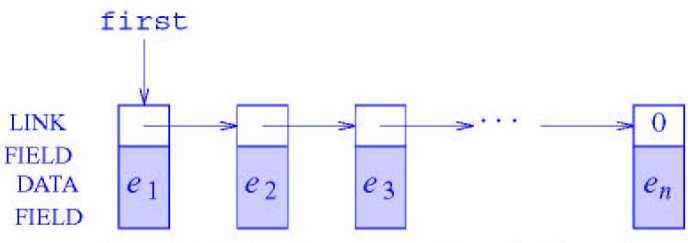
(a)



(b)

Εναλλακτικοί τρόποι αναπαράστασης

- Συνδεδεμένη αναπαράσταση
  - Κάθε στοιχείο της λίστας αναπαρίσταται σε ένα κελί (cell) ή κόμβο (node). Οι κόμβοι συνδέονται μεταξύ τους με συνδέσμους (links) ή δείκτες (pointers). Παράδειγμα:



  - Η δομή αυτή καλείται και αλυσίδα (chain)

Στατική κλάση Linear List

```
class LinearList {
   public:
      LinearList(int MaxListSize = 10); // constructor
      ~LinearList() {delete [] element;} // destructor
      bool IsEmpty() const {return length == 0;}
      int Length() const {return length;}
      bool Find(int k, T& x) const;
         // return the k'th element of list in x
      int Search(const T& x) const;
         // return position of x
      LinearList<T>& Delete(int k, T& x);
         // delete k'th element and return in x
      LinearList<T>& Insert(int k, const T& x);
         // insert x just after k'th element
      void Output(ostream& out) const;
   private:
      int length; int MaxSize;
      T *element; // dynamic 1D array
};
```


Δημιουργία λίστας

```
LinearList<T>::LinearList(int MaxListSize)
{// Constructor for formula-based linear list.
   MaxSize = MaxListSize;
   element = new T[MaxSize];
   length = 0;
}
```

Εύρεση k-οστού στοιχείου (find)

```
bool LinearList<T>::Find(int k, T& x) const
{// Set x to the k'th element of the list.
 // Return false if no k'th; true otherwise.
   if (k < 1 || k > length) return false; // no k'th
   x = element[k - 1];
   return true;
}
```

Αναζήτηση στοιχείου x σε λίστα (search)

```
int LinearList<T>::Search(const T& x) const
{// Locate x.  Return position of x if found.
 // Return 0 if x not in list.
   for (int i = 0; i < length; i++)
      if (element[i] == x) return ++i;
   return 0;
 }
```


Διαγραφή k-οστού στοιχείου (delete)

```
LinearList<T>& LinearList<T>::Delete(int k, T& x)
{// Set x to the k'th element and delete it.
 // Throw OutOfBounds exception if no k'th element.
   if (Find(k, x)) {// move elements k+1, ..., down
      for (int i = k; i < length; i++)
         element[i-1] = element[i];
      length--;
      return *this;
      }
   else throw OutOfBounds();
}
```

Εισαγωγή νέου στοιχείου μετά το k-οστό (insert)

```
LinearList<T>& LinearList<T>::Insert(int k, const T& x)
{// Insert x after the k'th element.
 // Throw OutOfBounds exception if no k'th element.
 // Throw NoMem exception if list is already full.
   if (k < 0 || k > length) throw OutOfBounds();
   if (length == MaxSize) throw NoMem();
   // move one up
   for (int i = length-1; i >= k; i--)
      element[i+1] = element[i];
   element[k] = x;
   length++;
   return *this;
}
```


Εισαγωγή λίστας στη ροή εξόδου (output)

```
template<class T>
void LinearList<T>::Output(ostream& out)
{// Put the list into the stream out.
   for (int i = 0; i <= length; i++)
       out << element[i] << "   ";
}
// overload <<
template<class T>
ostream& operator<< (osteram& out, const LinearList<T>&
   x)
{
   x.Output(out);
   return out;
}
```

Παράδειγμα χρήσης

```
#include <iostream.h>
#include "llist.h"
#include "xcept.h"
void main(void) {
   try {
        LinearList<int> L(5);
        cout << "Length=" << L.Length() << endl;
        cout << "IsEmpty=" << L.IsEmpty() << endl;
        L.Insert(0,2).Insert(1,6);
        cout << "List is" << L << endl;
        cout << "IsEmpty=" << L.IsEmpty() << endl;
        int z;
        L.Find(1,z);
        cout << "First element is" << z << endl;
        cout << "Length=" << L.Length() << endl;
        L.Delete(1,z);
        cout << "Deleted element is" << z << endl;
        cout << "List is" << L << endl;
        }
   catch(...) {
        cerr << "An exception has occurred" << endl;
        }
}
```


Συνδεδεμένη αναπαράσταση: κλάση Chain

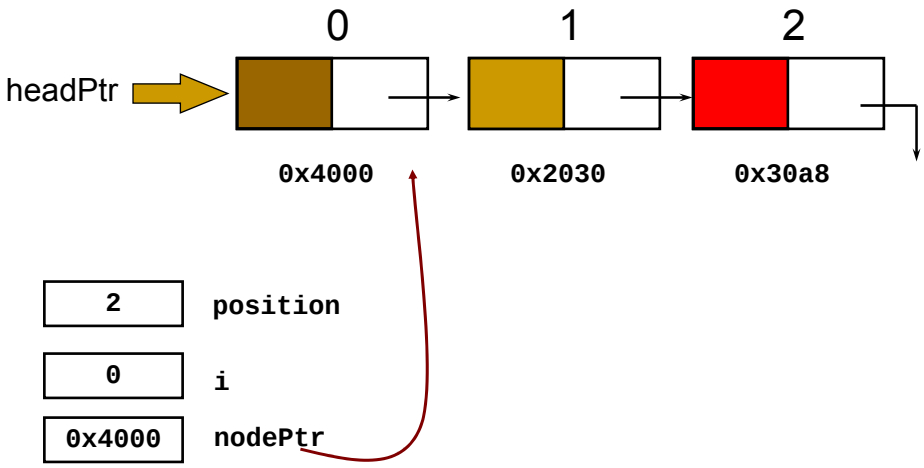
```
class Chain {
   public:
      Chain() {first = 0;}
      ~Chain();
      bool IsEmpty() const
          {return first == 0;}
      int Length() const;
      bool Find(int k, T& x) const;
      int Search(const T& x) const;
      Chain<T>& Delete(int k, T& x);
      Chain<T>& Insert(int k, const T& x);
      void Output(ostream& out) const;
   private:
      ChainNode<T> *first;  // pointer to first node
};
```

```
class ChainNode {
   friend Chain<T>;
  private:
      T data;
      ChainNode<T>
  *link;
};
```

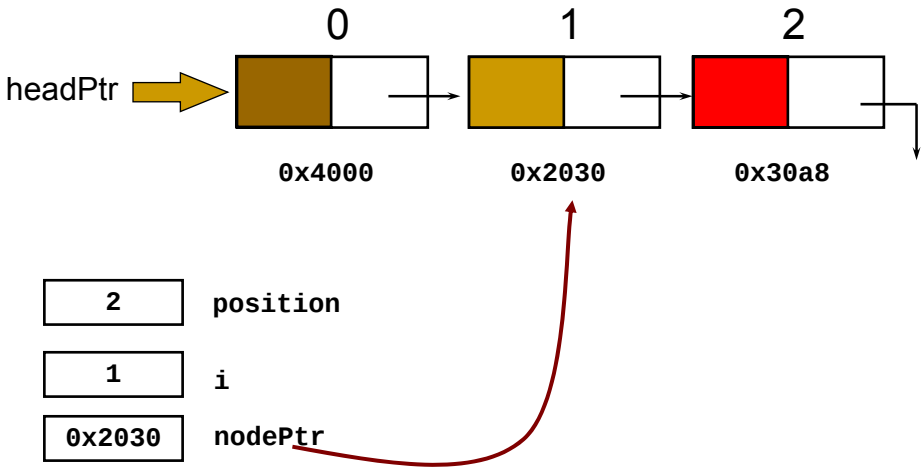
Τοποθέτηση σε συγκεκριμένη θέση

- Έλεγχος αν η θέση (position) είναι μέσα στα όρια.
- Αρχίζουμε από τον κόμβο head
- Θέτουμε index = 0
- while index < position
  - Ακολουθούμε το δείκτη next
  - Αυξάνουμε τη μεταβλητή index
- Επιστρέφουμε τον κόμβο

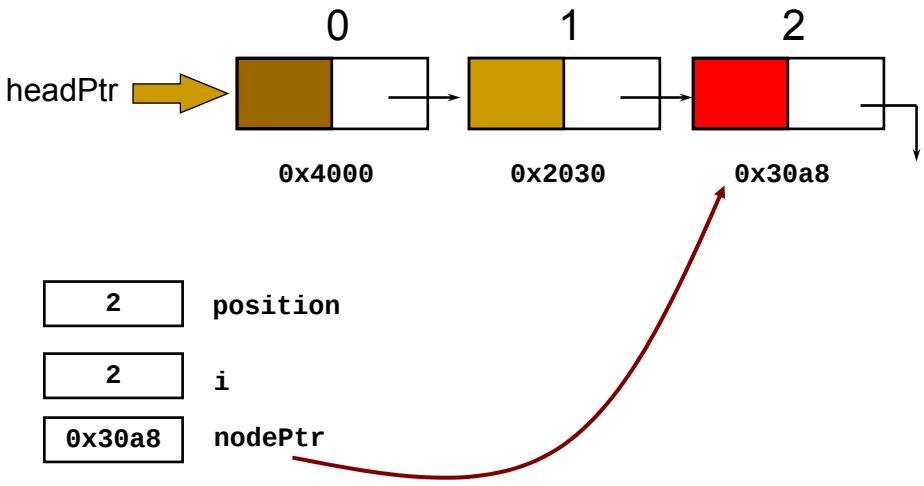
Τοποθέτηση σε συγκεκριμένη θέση



Τοποθέτηση σε συγκεκριμένη θέση



Τοποθέτηση σε συγκεκριμένη θέση



Εύρεση k-οστού στοιχείου αλυσίδας (find)

```
bool Chain<T>::Find(int k, T& x) const
{// Set x to the k'th element in the chain.
 // Return false if no k'th; return true otherwise.
   if (k < 1) return false;
   ChainNode<T> *current = first;
   int index = 1;  // index of current
   while (index < k && current) {
      current = current->link;
      index++;
      }
   if (current) {
      x = current->data; return true;}
   return false; // no k'th element
}
```

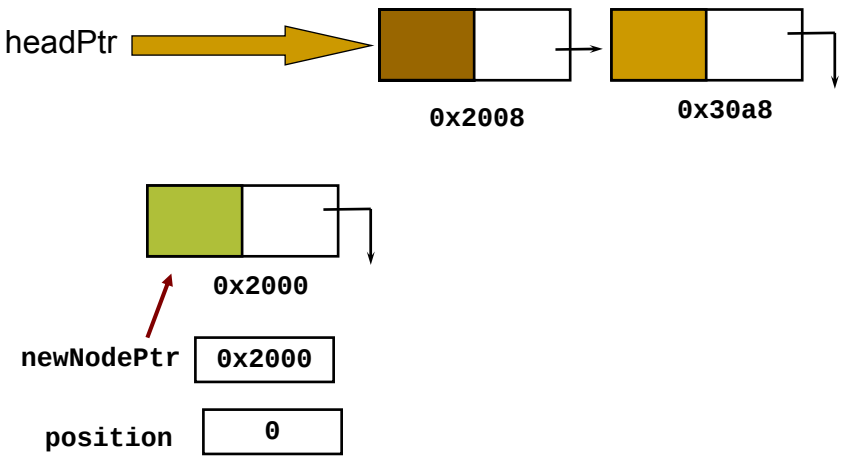

Αναζήτηση στοιχείου x σε αλυσίδα (search)

```
int Chain<T>::Search(const T& x) const
{// Locate x.  Return position of x if found.
 // Return 0 if x not in the chain.
   ChainNode<T> *current = first;
   int index = 1;  // index of current
   while (current && current->data != x) {
      current = current->link;
      index++;
      }
   if (current) return index;
   return 0;
}
```

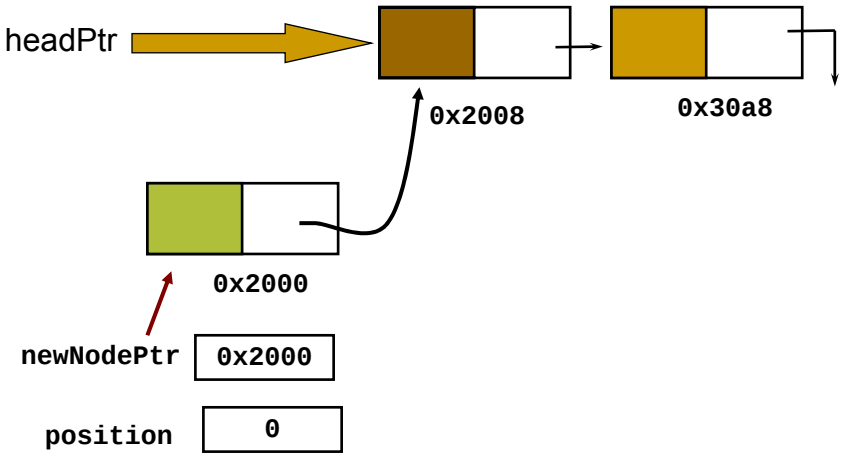
Προσδιορισμός μήκους αλυσίδας (length)

```
int Chain<T>::Length() const
{// Return the number of elements in the chain.
   ChainNode<T> *current = first;
   int len = 0;
   while (current) {
     len++;
     current = current->link;
     }
   return len;
}
```

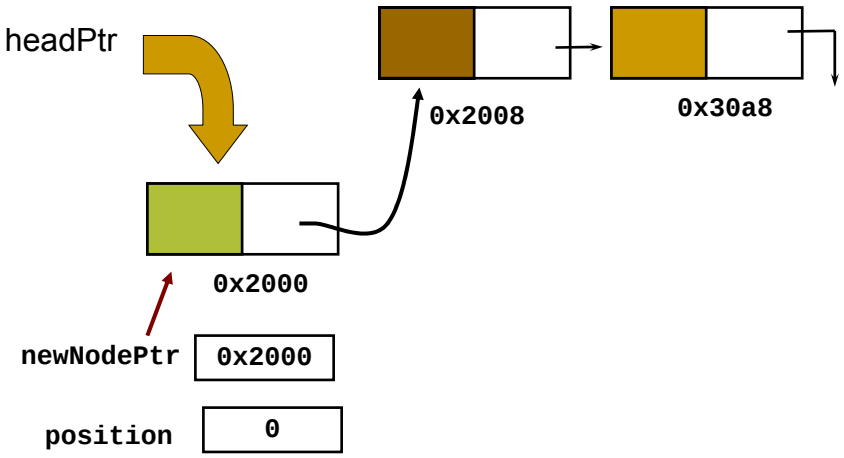

Εισαγωγή στη Αρχή



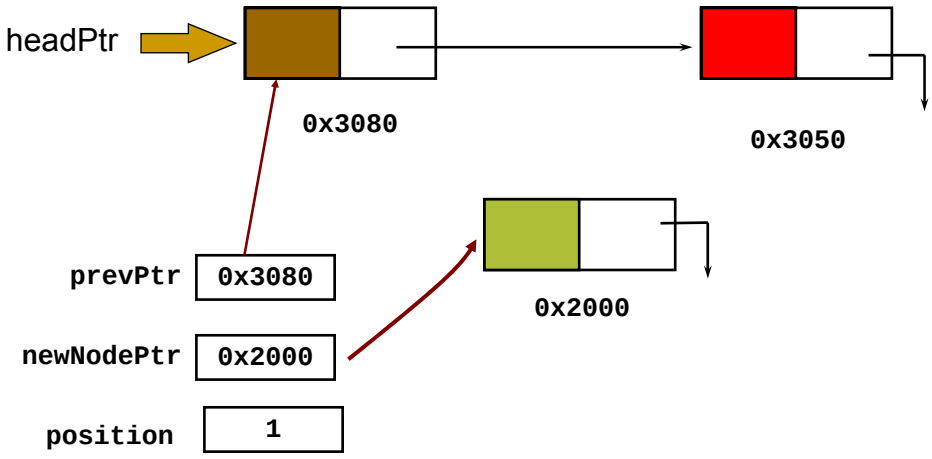
Εισαγωγή στην Αρχή



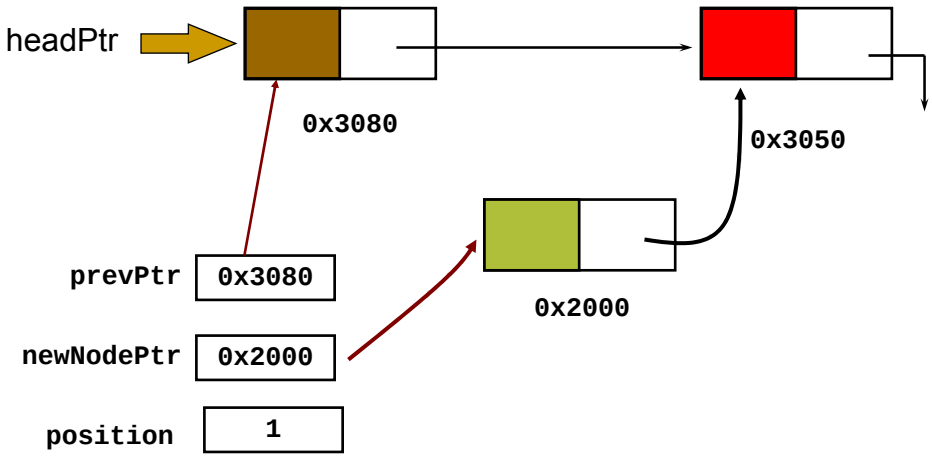
Εισαγωγή στην Αρχή



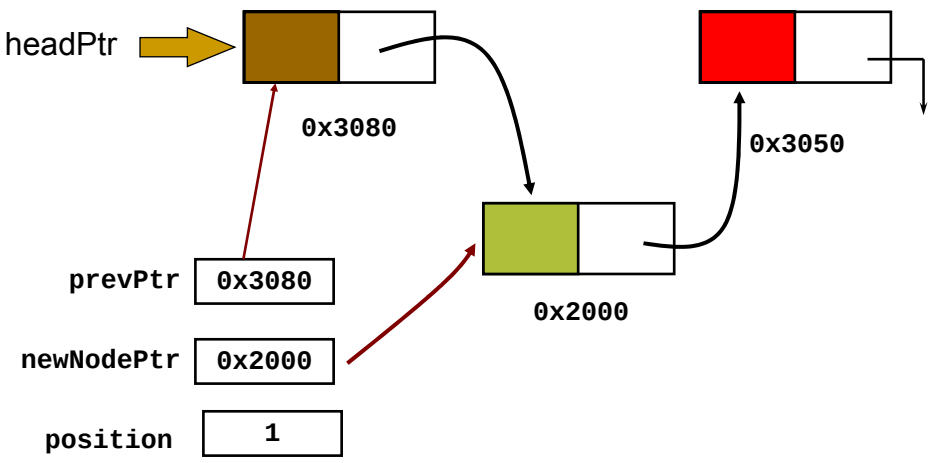
Εισαγωγή Ενδιάμεσα



Εισαγωγή Ενδιάμεσα



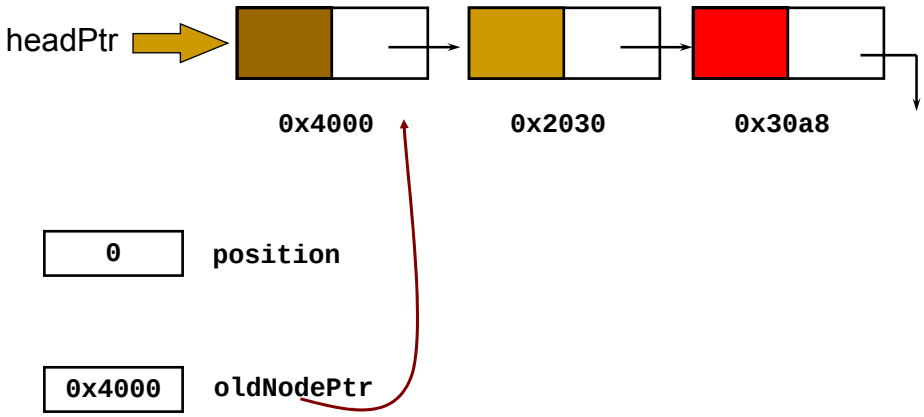
Εισαγωγή Ενδιάμεσα



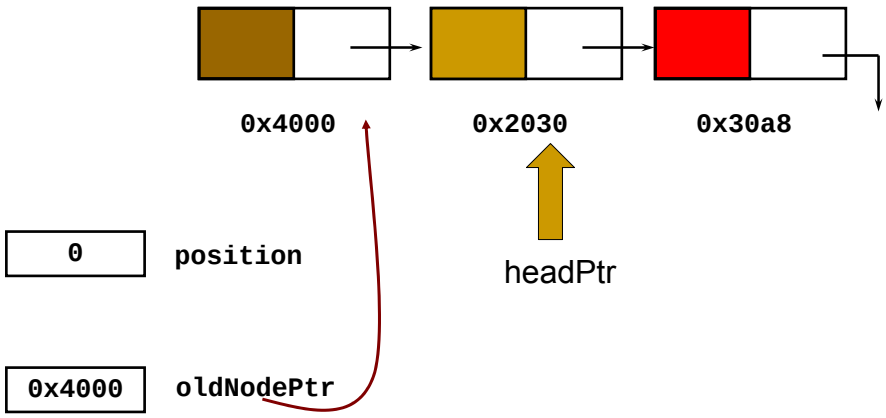
Εισαγωγή στοιχείου σε αλυσίδα

```
Chain<T>& Chain<T>::Insert(int k, const T& x)
{// Insert x after the k'th element.
 // Throw OutOfBounds exception if no k'th element.
 // Pass NoMem exception if inadequate space.
   if (k < 0) throw OutOfBounds();
   // p will eventually point to k'th node
   ChainNode<T> *p = first;
   for (int index = 1; Index < k && p; index++)
      p = p->link; // move p to k'th
   if (k > 0 && !p) throw OutOfBounds(); // no k'th
   // insert
   ChainNode<T> *y = new ChainNode<T>; y->data = x;
   if (k) {// insert after p
           y->link = p->link; p->link = y;}
   else {// insert as first element
         y->link = first; first = y;}
   return *this;
}
```

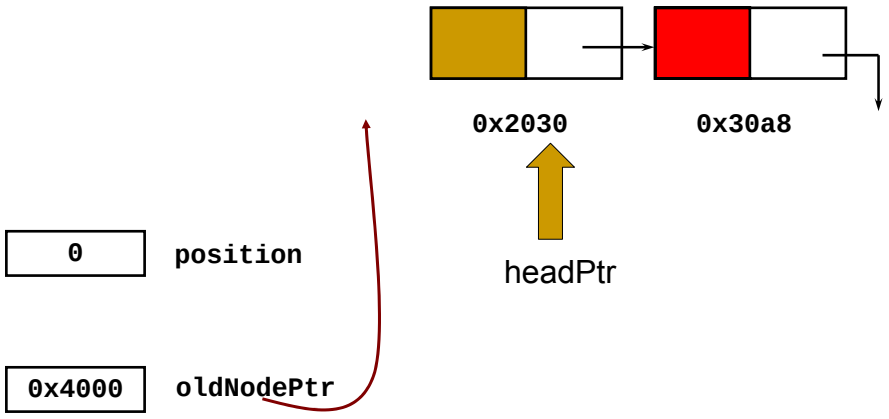
Διαγραφή 1ου Κόμβου



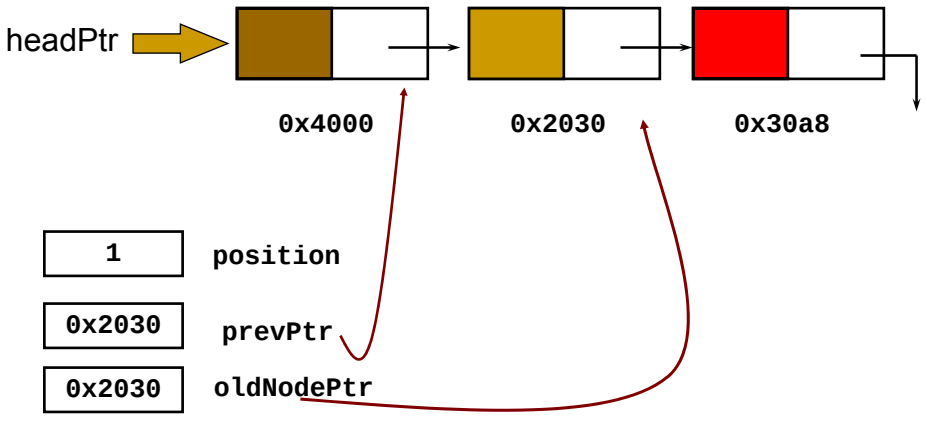
Διαγραφή 1ου Κόμβου



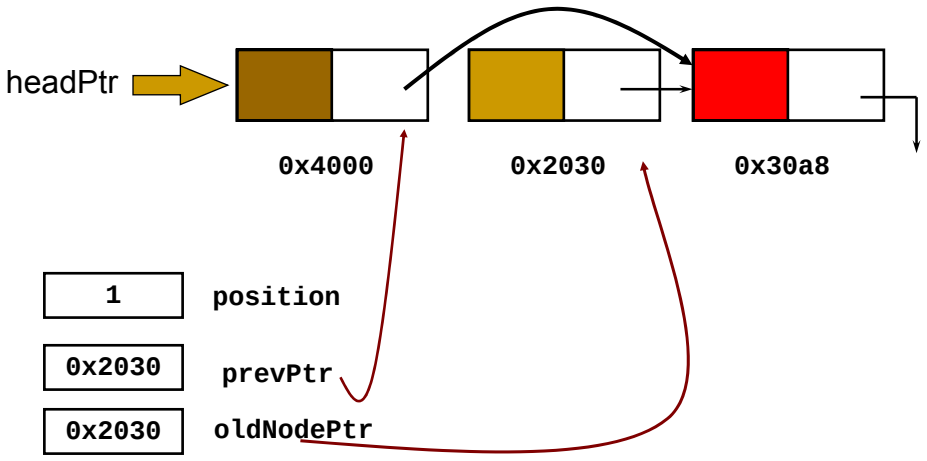
Διαγραφή 1ου Κόμβου



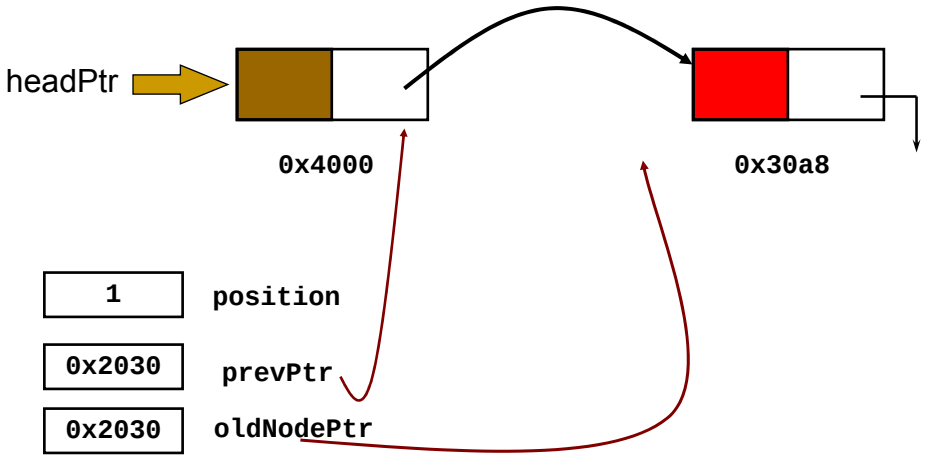
Διαγραφή Ενδιάμεσου Κόμβου



Διαγραφή Ενδιάμεσου Κόμβου



Διαγραφή Ενδιάμεσου Κόμβου



Διαγραφή στοιχείου από αλυσίδα

```
Chain<T>& Chain<T>::Delete(int k, T& x)
{// Set x to the k'th element and delete it.
 // Throw OutOfBounds exception if no k'th element.
   if (k < 1 || !first) throw OutOfBounds(); // no k'th
   // p will eventually point to k'th node
   ChainNode<T> *p = first;
   // move p to k'th & remove from chain
   if (k == 1) // p already at k'th
      first = first->link; // remove
   else { // use q to get to k-1'st
      ChainNode<T> *q = first;
      for (int index = 1; index < k - 1 && q; index++)
         q = q->link;
      if (!q || !q->link) throw OutOfBounds(); // no k'th
      p = q->link; q->link = p->link;} // remove k'th
   // save k'th element and free node p
   x = p->data; delete p; return *this;
}
```


Διαγραφή όλων των κόμβων της αλυσίδας (συνάρτηση καταστροφής – destructor)

```
Chain<T>::~Chain()
{// Chain destructor. Delete all nodes in chain.
   ChainNode<T> *next;  // next node
   while (first) {
      next = first->link;
      delete first;
      first = next;
      }
 }
```

Σύγκριση των υλοποιήσεων

Στατικές λίστες (υλοποίηση με πίνακα):

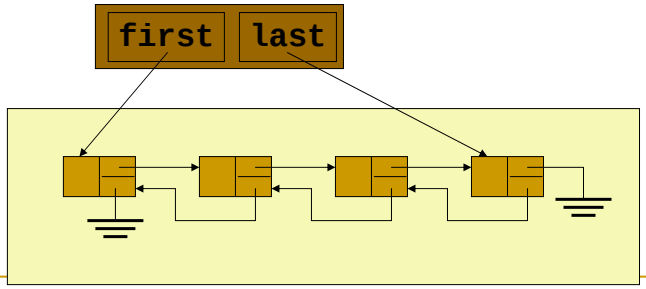
- Εισαγωγή και διαγραφή k-οστού στοιχείου είναι $\Theta(n)$
- Εύρεση τιμής k-οστού στοιχείου είναι $\Theta(1)$
- Αναζήτηση στοιχείου με τιμή x είναι $\Theta(n)$
- Ο χώρος πρέπει να δεσμευτεί εκ των προτέρων

Συνδεδεμένες λίστες (αλυσίδες):

- Εισαγωγή και διαγραφή k-οστού στοιχείου είναι $\Theta(n)$
- Εύρεση τιμής k-οστού στοιχείου είναι $\Theta(n)$
- Αναζήτηση στοιχείου με τιμή x είναι $\Theta(n)$
- Ο χώρος δεσμεύεται δυναμικά με τον αριθμό των στοιχείων

Διπλά Συνδεδεμένη Λίστα

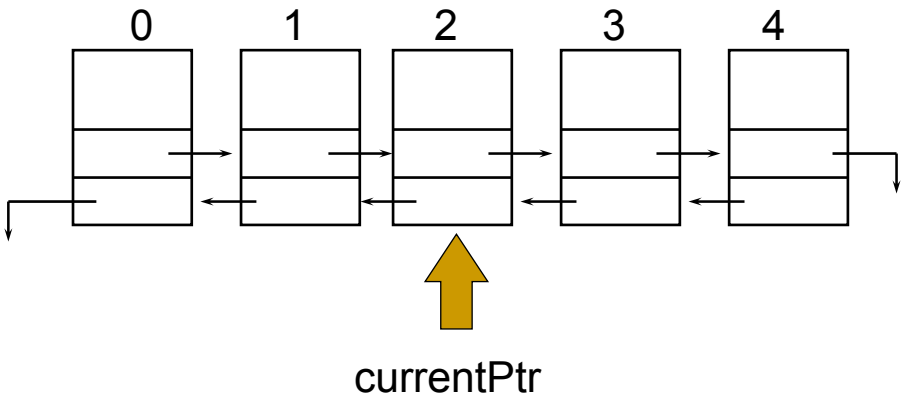
- Επίσης γραμμικές διατάξεις
- Δυο σύνδεσμοι σε κάθε κόμβο, προς τον επόμενο και προς τον προηγούμενο
- Γενική μορφή, π.χ. για υλοποίηση ουράς:



Διπλά Συνδεδεμένη Λίστα

- Πήγαινε σε συγκεκριμένη θέση
- Εισαγωγή σε συγκεκριμένη θέση
- Διαγραφή από συγκεκριμένη θέση
- Ανάγνωση δεδομένων συγκεκριμένης θέσης
- Αντικατάσταση
- Διάσχιση λίστας ***και στις δύο κατευθύνσεις***.

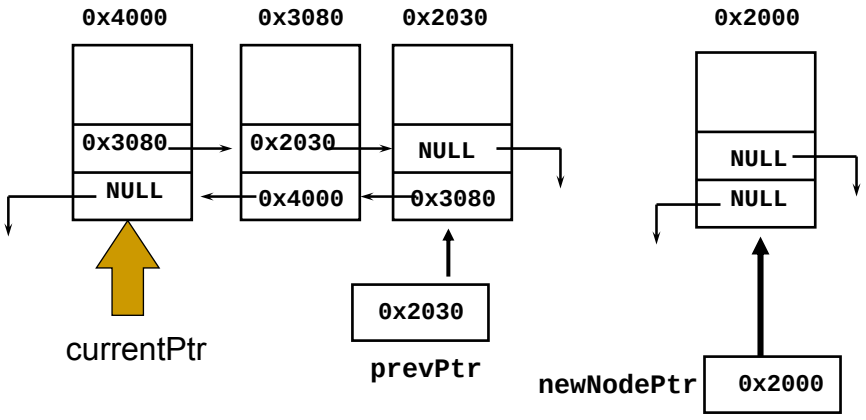
Διπλά Συνδεδεμένη Λίστα



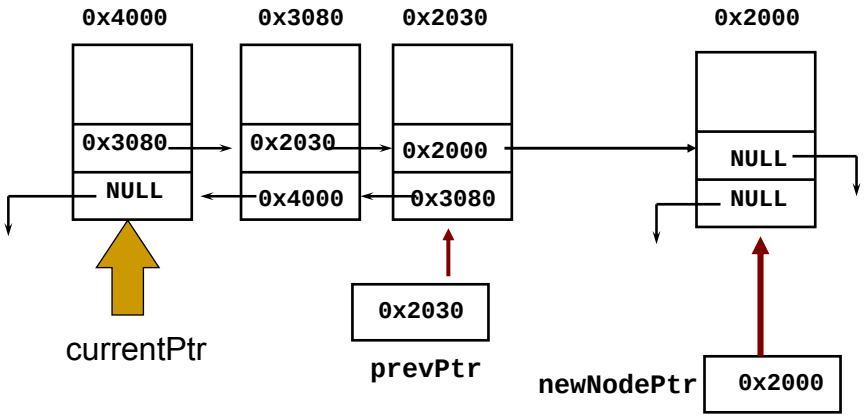
Διπλά Συνδεδεμένη Λίστα

```
struct DoubleLinkNodeRec
{
   float                        value;
   struct DoubleLinkNodeRec*    nextPtr;
   struct DoubleLinkNodeRec*    previousPtr;
};

typedef struct DoubleLinkNodeRec Node;

struct DoubleLinkListRec
{
   int      count;
   Node*    currentPtr;
   int      position;
};

typedef struct DoubleLinkListRec DoubleLinkList;
```

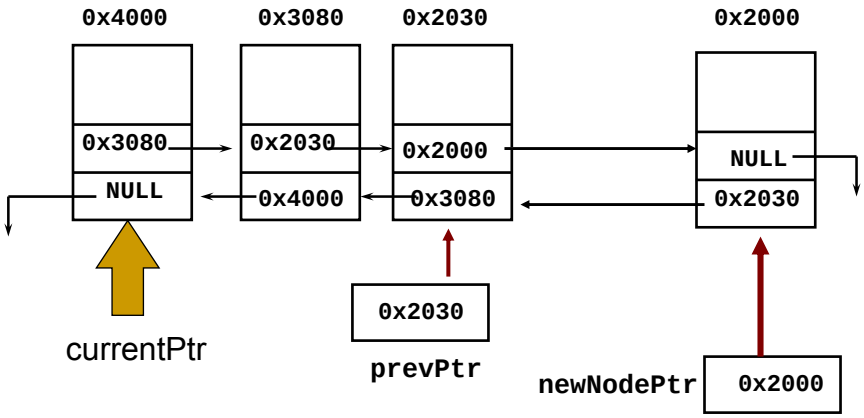

Εισαγωγή στο Τέλος



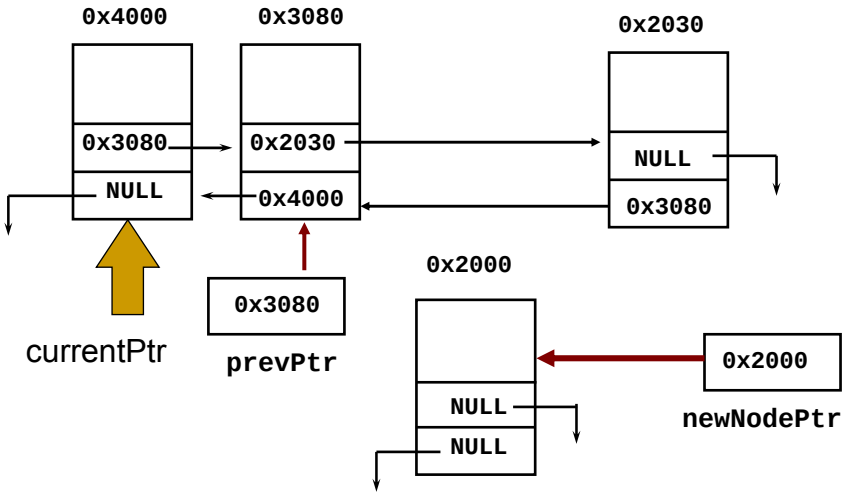
Εισαγωγή στο Τέλος



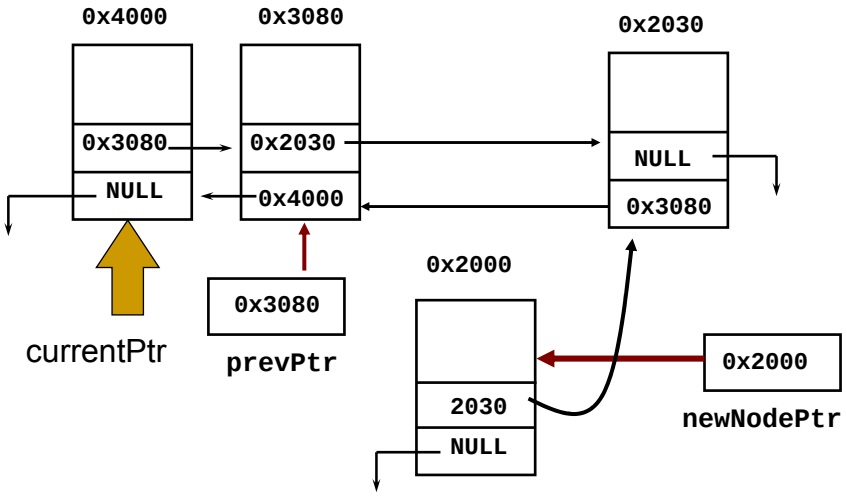
Εισαγωγή στο Τέλος



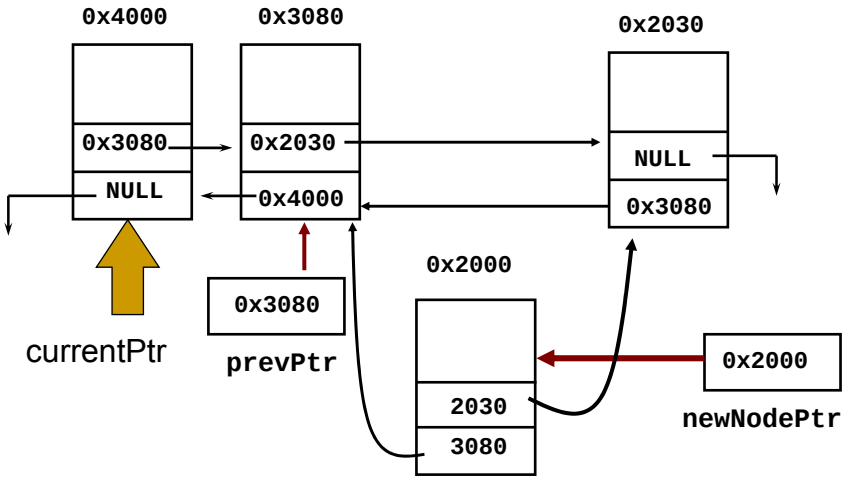
Εισαγωγή Ενδιάμεσα



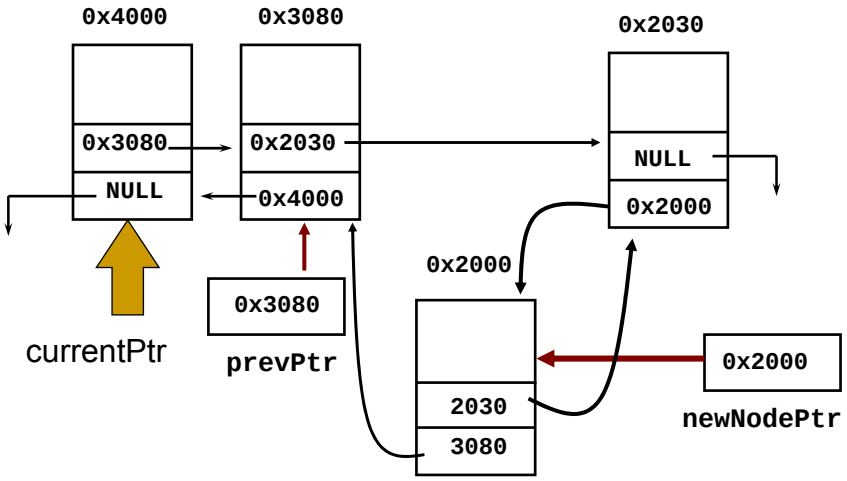
Εισαγωγή Ενδιάμεσα



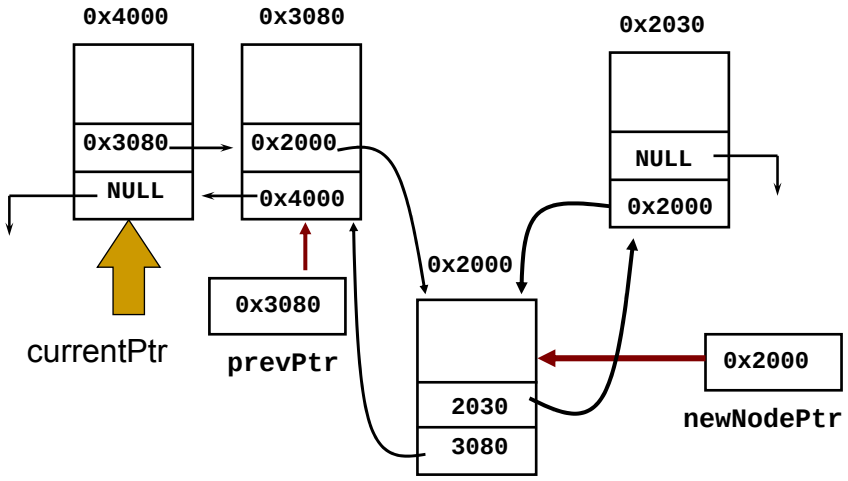
Εισαγωγή Ενδιάμεσα



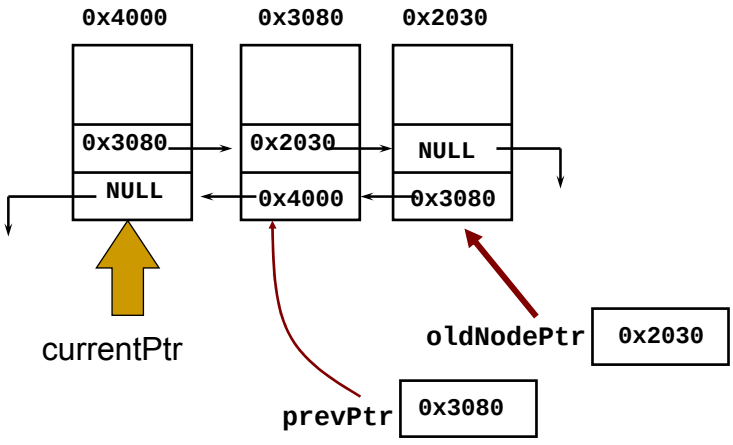
Εισαγωγή Ενδιάμεσα



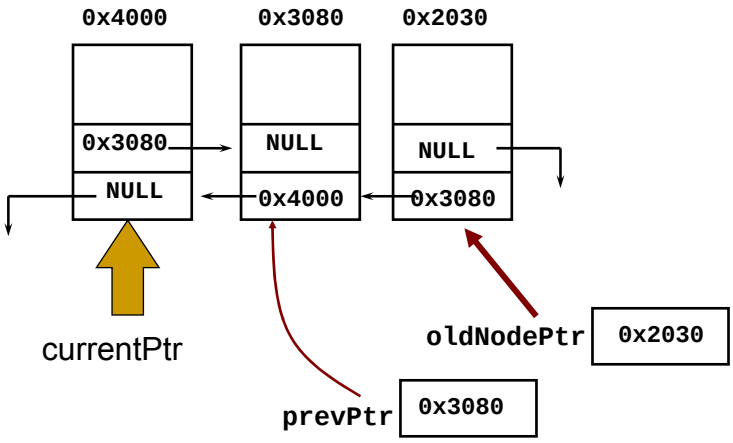
Εισαγωγή Ενδιάμεσα



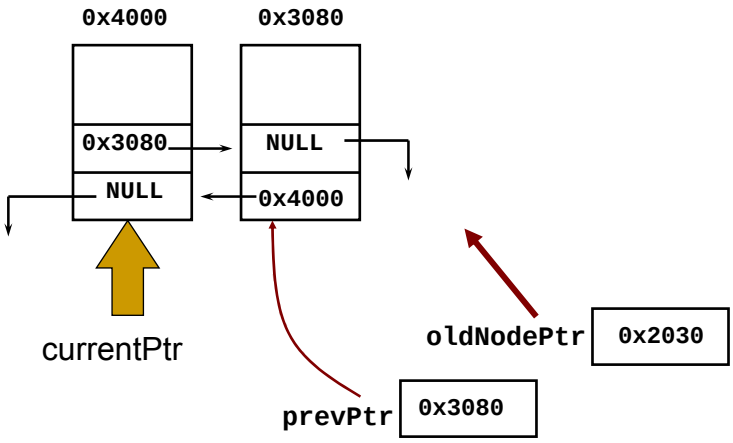
Διαγραφή από το Τέλος



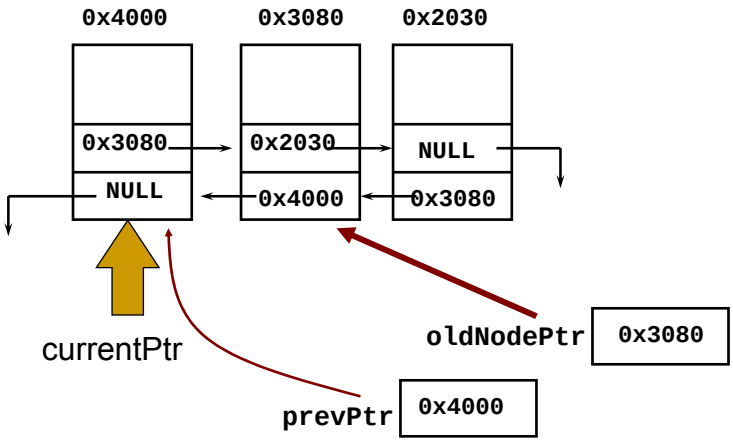
Διαγραφή από το Τέλος



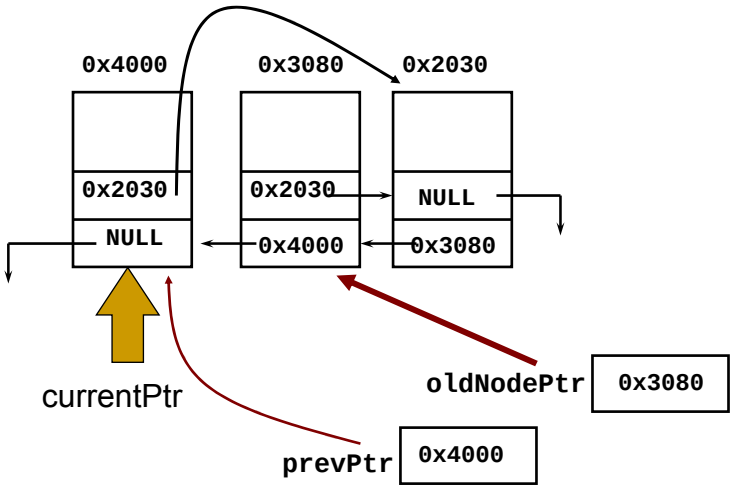
Διαγραφή από το Τέλος



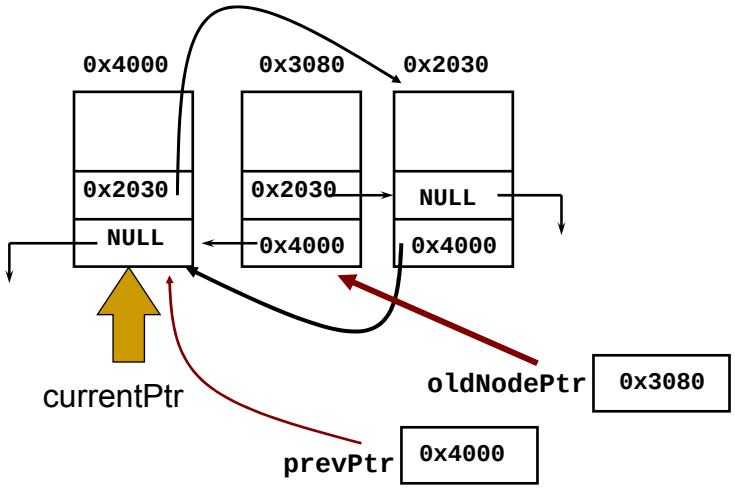
Διαγραφή Ενδιάμεσου



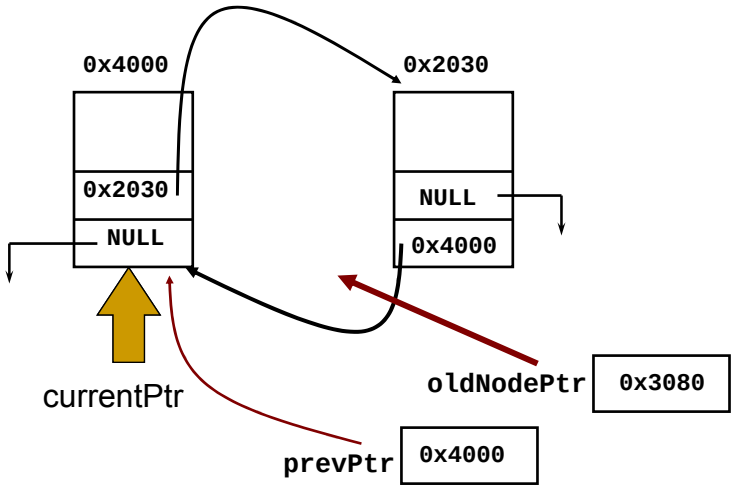
Διαγραφή Ενδιάμεσου



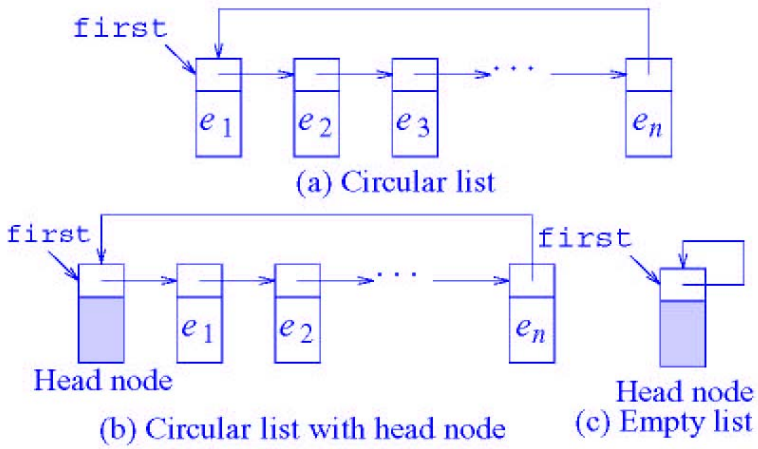
Διαγραφή Ενδιάμεσου



Διαγραφή Ενδιάμεσου



Κυκλικά συνδεδεμένες λίστες



Κυκλικά συνδεδεμένες λίστες

```
int CircularList<T>::Search(const T& x) const
{// Locate x in a circular list with head node.
   ChainNode<T> *current = first->link;
   int index = 1; // index of current
   first->data = x; // put x in head node

   // search for x
   while (current->data != x) {
      current = current->link);
      index++;
      }

   // are we at head?
   return ((current == first) ? 0 : index);
}
```