Pontificia Universidad
JAVERIANA
Colombia

Estructuras de Datos

List Data Structure

In This Session

- **List Data Structure**
  - ▷ Array-based Representation
  - ▷ Linked Representation

Data Structures
– in general –

A **data structure** is a data object along with the relationship that exist among the instances and elements, and which are provided by specifying the functions of interest.

Our *main* concern will be:

- The representation of data objects (actually of their instances)
- The representation should facilitate an *efficient* implementation of functions

List Data Structure

A *list* is a data object whose instances are of the form $(e_1, e_2, \ldots, e_n)$ where n is a finite natural number.

The e_i terms are the elements of the list and n is the length.

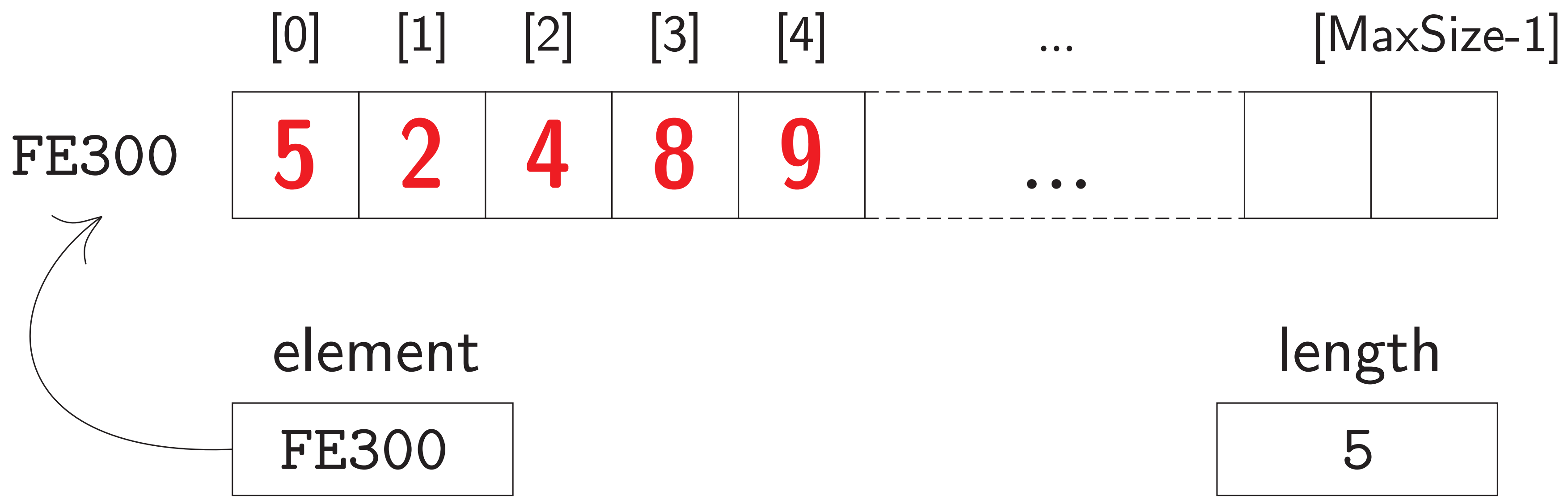
There exist different methods for representing a List, namely:

- Array-based Representation
- Linked Representation
- Indirect Addressing Representation
- Simulating Pointers Representation

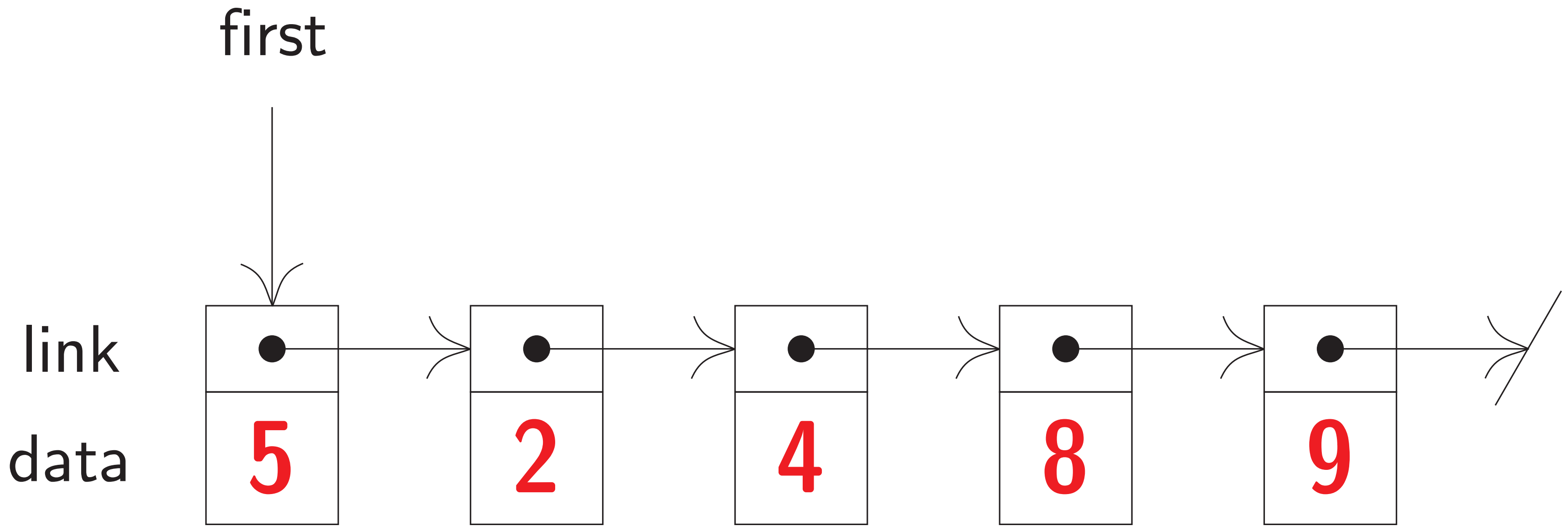
In this course we will only focus our attention on the first two representations

Methods of Representing a List

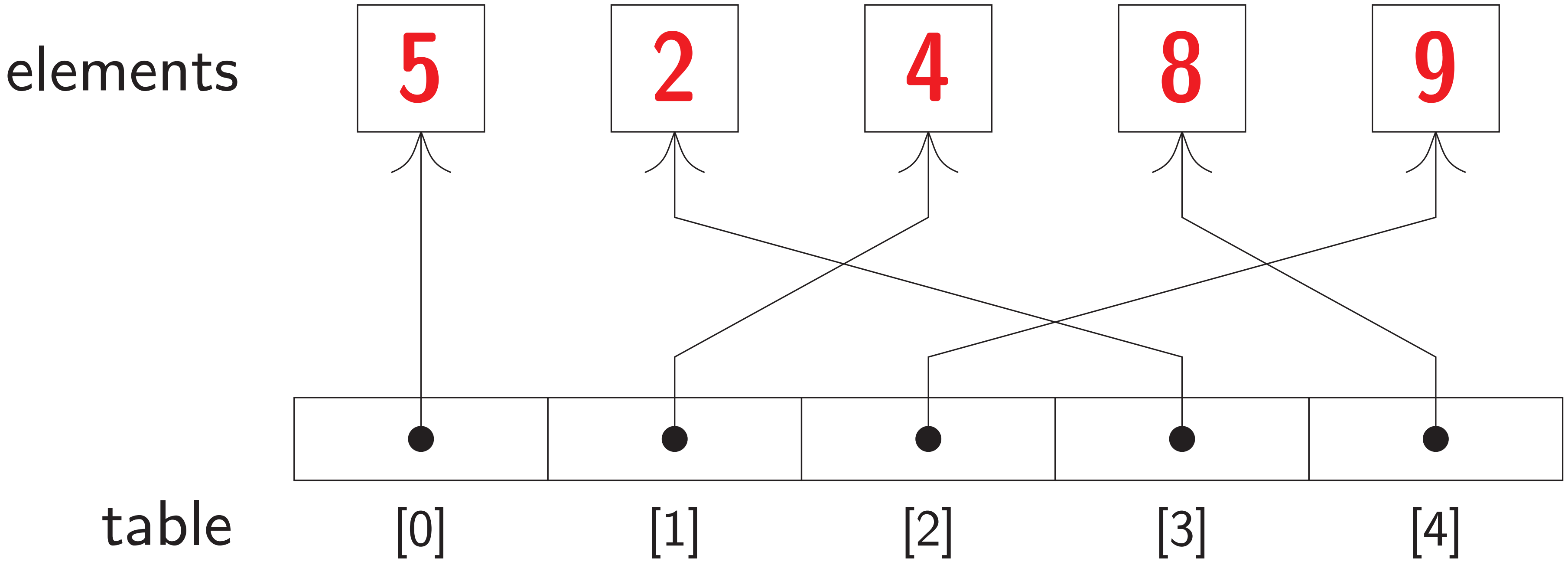
Array-based Representation: Uses a mathematical formula to determine where (i.e., the memory address) to store each element of a data object.



Linked Representation: The elements may be stored in any arbitrary set of memory locations. Each element keeps explicit information about the location of the next element called pointer (or link).



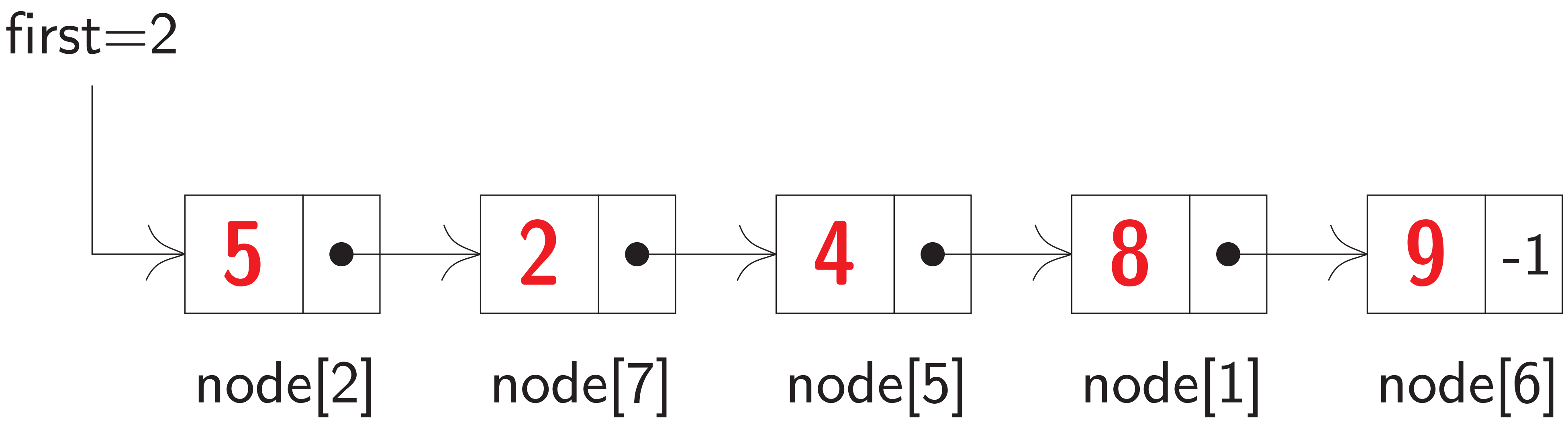
Indirect Addressing Representation: Is a combination of formula-based and linked representation where the addresses of the elements are collected into a table.



Simulating Pointers Representation: Similar to the linked list representation. However, pointers are replaced by integers.

firstNode $\boxed{2}$

node	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
next		6	7			1	−1	5		
element		8	5			4	9	2		



LinearList Data Structure

– Array-Based Representation –

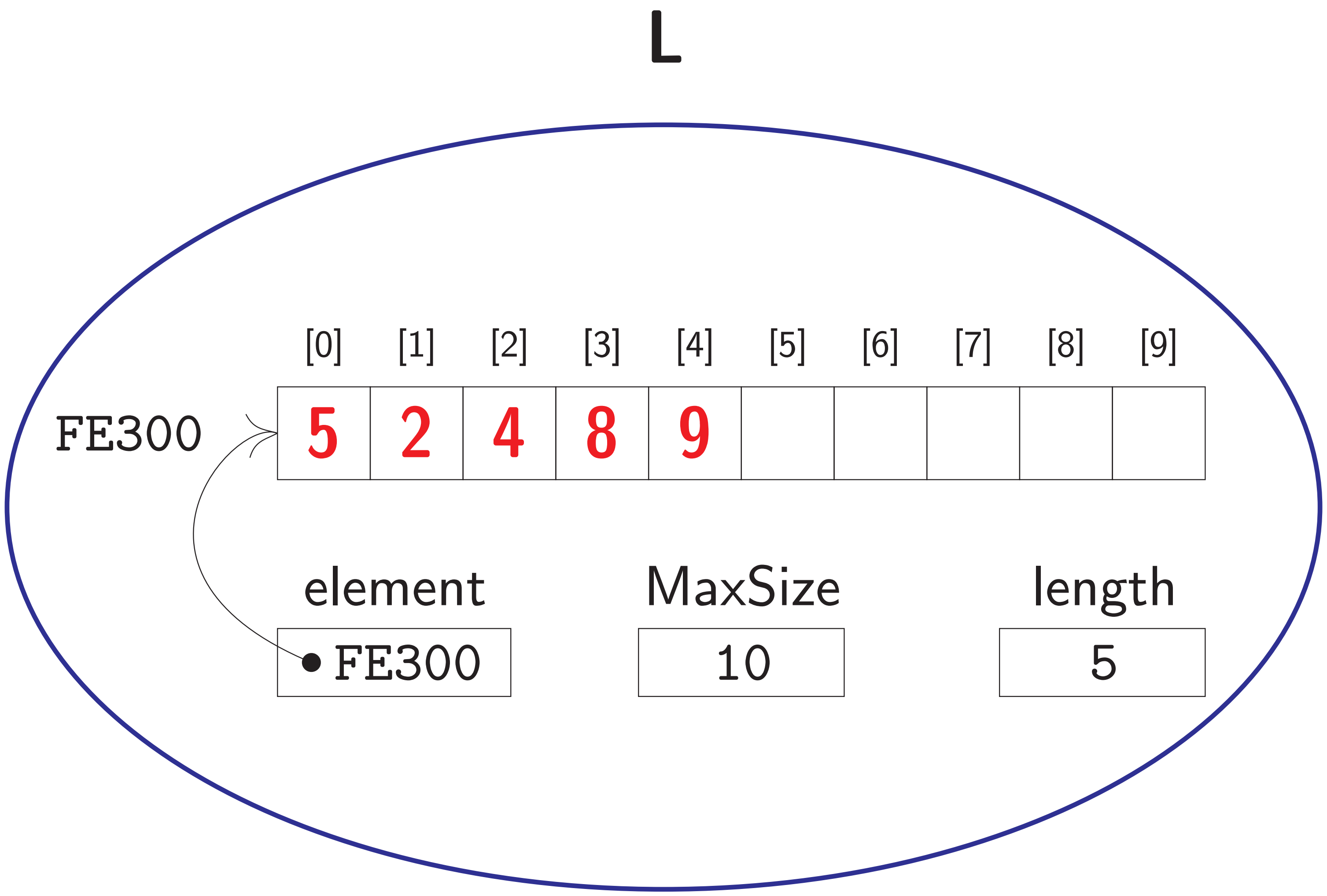
A formula-based representation uses an *array* to represent the instances of a linearlist. A simple formula is used to determine the location of each element:

$$location(i) = i - 1$$

This means that the ith element of the list (if it exists) is at position $i - 1$ of the array.

Example

An array list with 5 elements and max size 10 will look like this:



arraylist.h

```c
#ifndef ARRAYLIST
#define ARRAYLIST

#include <stdio.h>
#include <stdlib.h>
#define CAP 100

typedef struct List List;

struct List{
   int a[ CAP ], n;
   void (*add) ( List *, int, int );
   void (*delete) ( List *, int );
   int (*get) ( List *, int );
   int (*search) ( List *, int );
   void (*display) ( List * );
   int (*size) ( List * );
   int (*empty) ( List * );
};

void add( List * x, int index, int e ){ ... }
void delete( List * x, int index ){ ... }
```

```
int get( List * x, int index ){ ... }
int search( List * x, int e ){ ... }
void display( List * x ){ ... }
int size( List * x ){ ... }
int empty( List * x ){ ... }
List createList( ){ ... }

#endif
```

createList

```
List createList( ){
   List l;
   l.n = 0;
   l.add = &add;
   l.delete = &delete;
   l.get = &get;
   l.search = &search;
   l.display = &display;
   l.size = &size;
   l.empty = &empty;
   return l;
}
```

add

```
void add( List * x, int index, int e ){
   int i;
   if( index < 0 || index > x->n ){
      fprintf( stderr, "Error␣:␣invalid␣index\n" );
      return;
   }
   if( x->n == CAP ){
      fprintf( stderr, "Error␣:␣list␣is␣full\n" );
      return;
   }
   for( i = x->n - 1; i >= index; i-- )
      x->a[ i + 1 ] = x->a[ i ];
   x->a[ index ] = e;
   x->n++;
   return;
}
```

delete

```
void delete( List * x, int index ){
   int i;
   if( index < 0 || index >= x->n ){
      fprintf( stderr, "Error␣:␣invalid␣index\n" );
      return;
   }
   for( i = index; i < x->n - 1; i++ )
      x->a[ i ] = x->a[ i + 1 ];
   x->n--;
   return;
}
```

get, search

```
int get( List * x, int index ){
   if( index < 0 || index >= x->n ){
      fprintf( stderr, "Error␣:␣invalid␣index\n" );
      return;
   }
   return x->a[ index ];
}


int search( List * x, int e ){
   int i;
   for( i = 0; i < x->n; i++ )
      if( x->a[ i ] == e )
         return i;
   return -1;
}
```

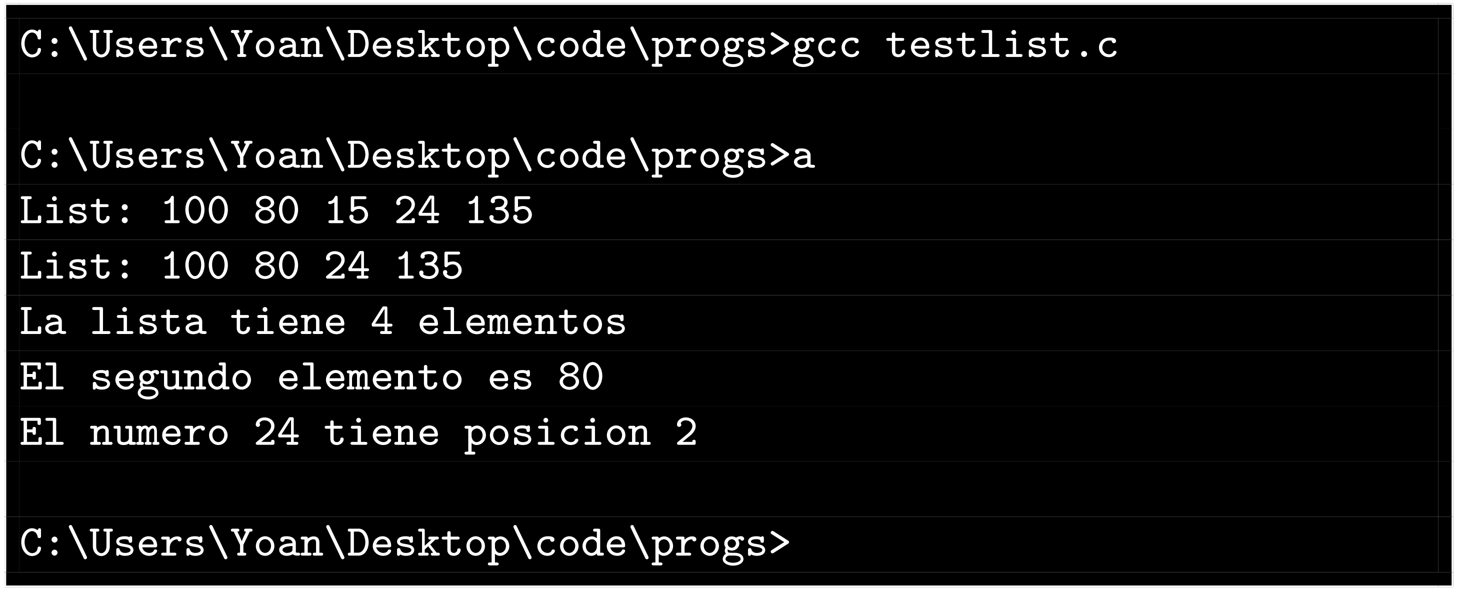
display, size, empty

```
void display( List * x ){
   int i;
   printf( "List:␣" );
   for( i = 0; i < x->n; i++ )
      printf( "%d␣", x->a[ i ] );
   printf( "\n" );
   return;
}


int size( List * x ){
   return x->n;
}


int empty( List * x ){
   return x->n == 0;
}
```

testlist.c

```c
#include <stdio.h>
#include "arraylist.h"

int main(){
   List maria = createList( );
   maria.add( &maria, 0, 100 );
   maria.add( &maria, 1, 80 );
   maria.add( &maria, 2, 15 );
   maria.add( &maria, 3, 24 );
   maria.add( &maria, 4, 135 );
   maria.display( &maria );
   maria.delete( &maria, 2 );
   maria.display( &maria );
   printf( "La␣lista␣tiene␣%d␣elementos\n", maria.size( &maria ) );
   printf( "El␣segundo␣elemento␣es␣%d\n", maria.get( &maria, 1 ) );
   printf( "El␣numero␣24␣tiene␣posicion␣%d\n", maria.search( &maria, 24 ) );
   return 0;
}
```

Compilation and Running



```
C:\Users\Yoan\Desktop\code\progs>gcc testlist.c

C:\Users\Yoan\Desktop\code\progs>a
List: 100 80 15 24 135
List: 100 80 24 135
La lista tiene 4 elementos
El segundo elemento es 80
El numero 24 tiene posicion 2

C:\Users\Yoan\Desktop\code\progs>
```

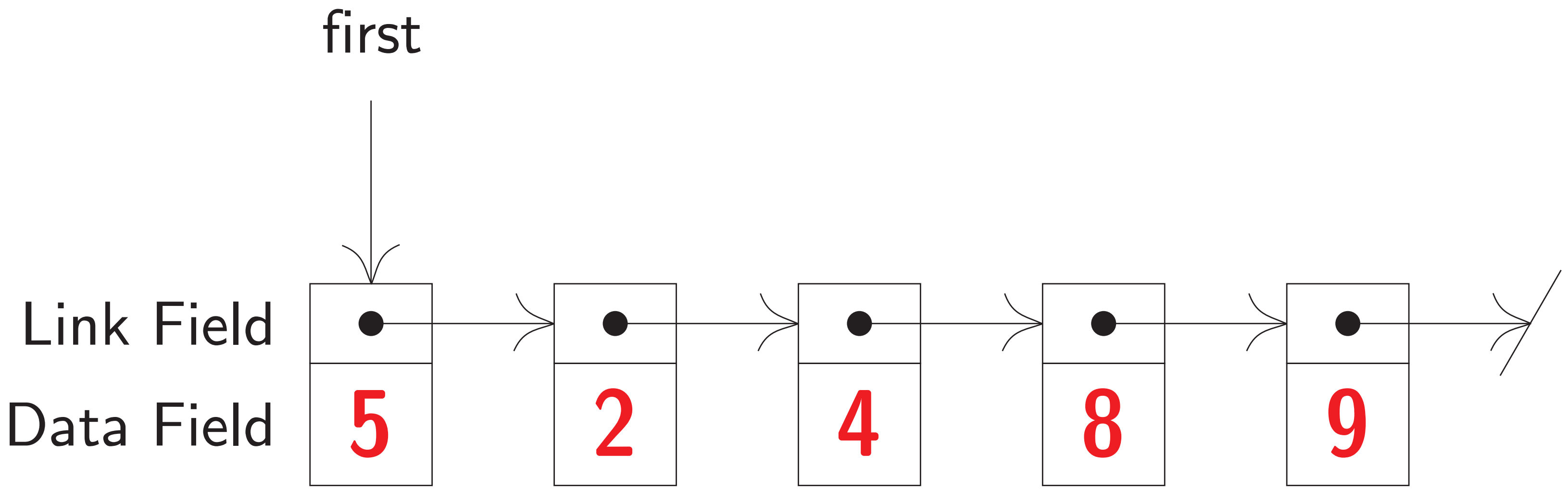
Complexity of Operations

add	:	$O(n)$
delete	:	$O(n))$
get	:	$\Theta(1)$
search	:	$O(n)$
display	:	$\Theta(n)$
size	:	$\Theta(1)$
empty	:	$\Theta(1)$
createList	:	$\Theta(1)$

LinearList Data Structure

– Linked Representation –

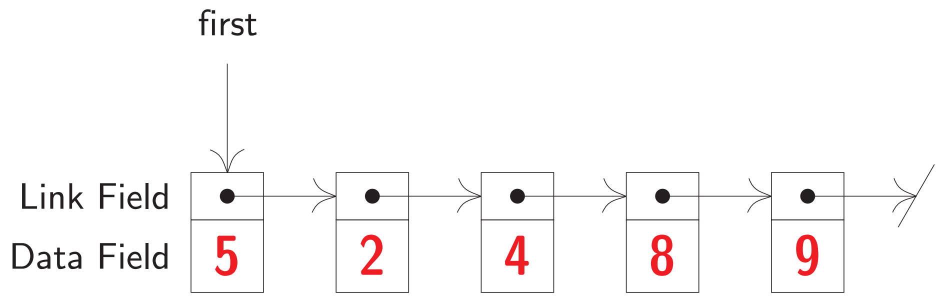
- Each element of an instance of a data object is represented as a group of memory locations called *cell* or *node*.
- The elements may be stored in any arbitrary set of memory locations.
- Each element keeps explicit information about the location of the next element through a *pointer* (or *link*)
- Also called *Chain*

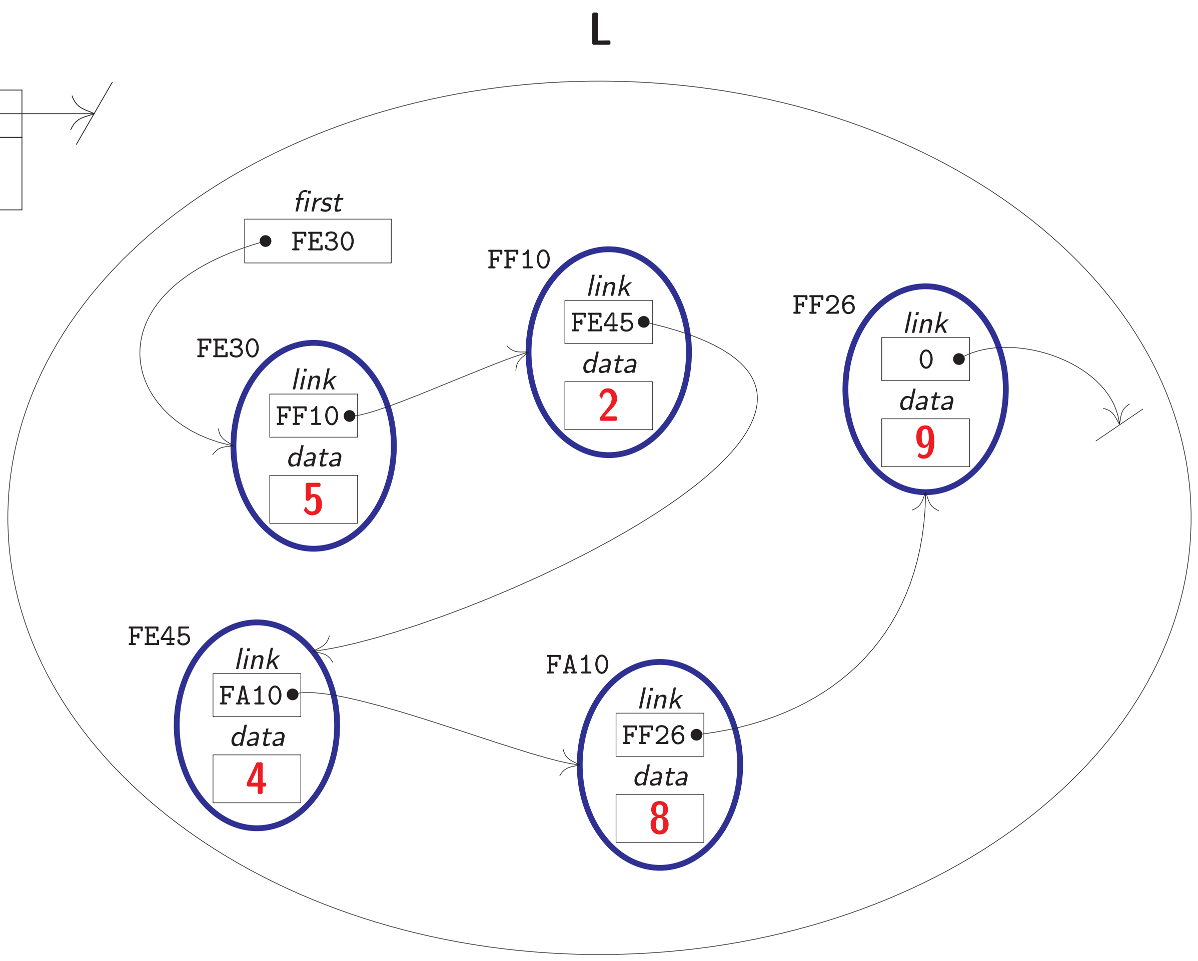


Singly linked list or chain

Example

A linked list with 5 elements will look like this:





linkedlist.h

```c
#ifndef LINKEDLIST
#define LINKEDLIST

#include <stdio.h>
#include <stdlib.h>

typedef struct List List;
typedef struct Node Node;

struct Node{
   int item;
   Node * next;
};

struct List{
   Node * firstNode;
   int n;
   void (*add) ( List *, int, int );
   void (*delete) ( List *, int );
   int (*get) ( List *, int );
   int (*search) ( List *, int );
   void (*display) ( List * );
   int (*size) ( List * );
   int (*empty) ( List * );
};
```

```c
void add( List * x, int index, int e ){ ... }
void delete( List * x, int index ){ ... }
int get( List * x, int index ){ ... }
int search( List * x, int e ){ ... }
void display( List * x ){ ... }
int size( List * x ){ ... }
int empty( List * x ){ ... }
List createList( ){ ... }

#endif
```

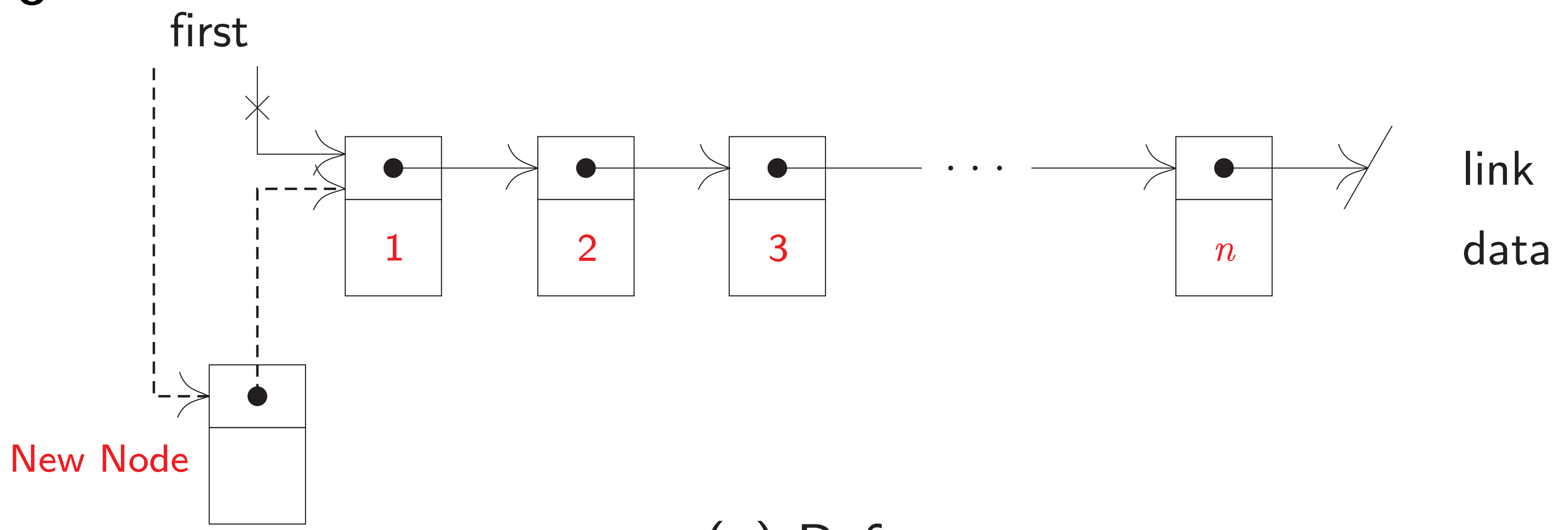
Important: this representation does not specify any capacity

createList

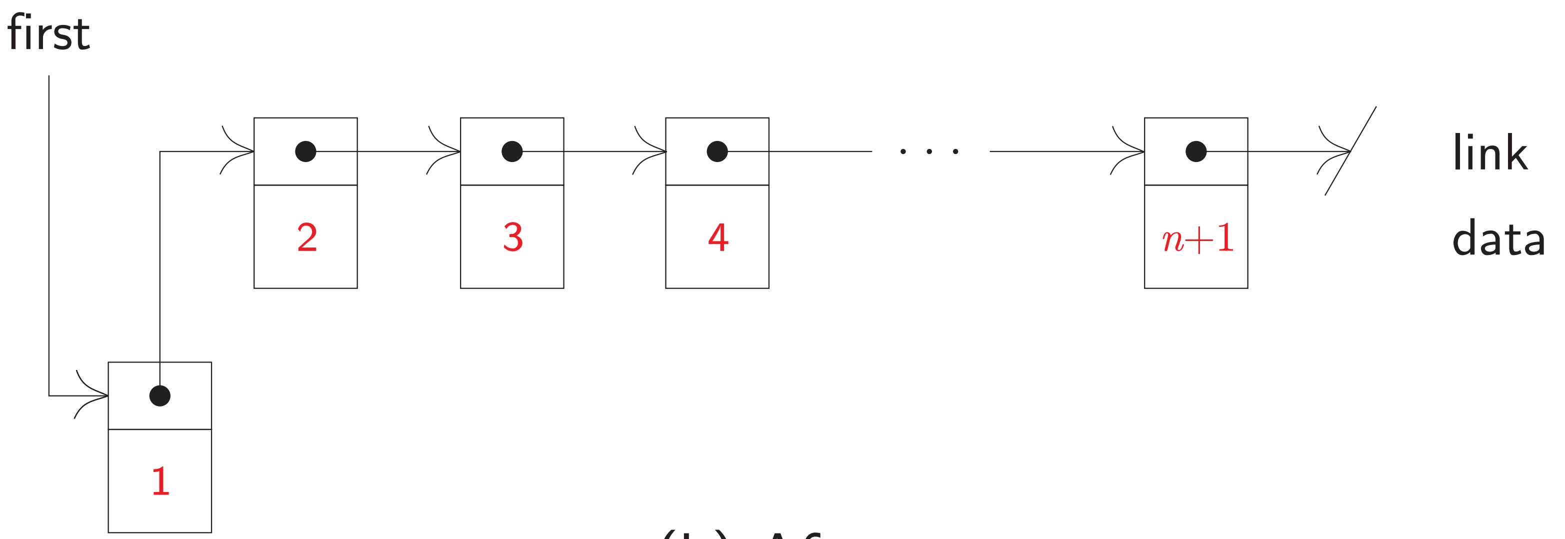
```
List createList( ){
   List l;
   l.firstNode = NULL;
   l.n = 0;
   l.add = &add;
   l.delete = &delete;
   l.get = &get;
   l.search = &search;
   l.display = &display;
   l.size = &size;
   l.empty = &empty;
   return l;
}
```

Adding after the kth element

If k=0

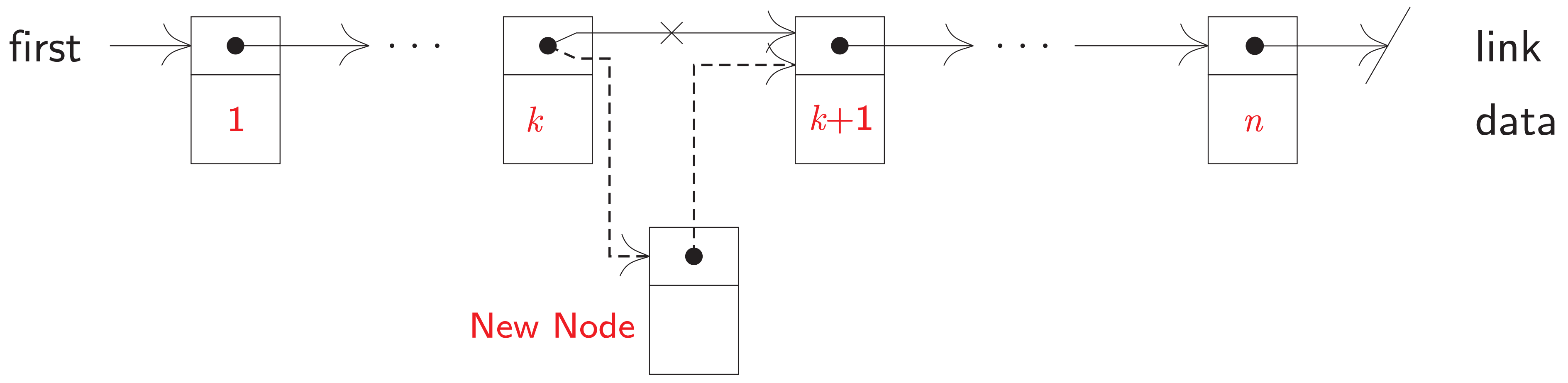


(a) Before

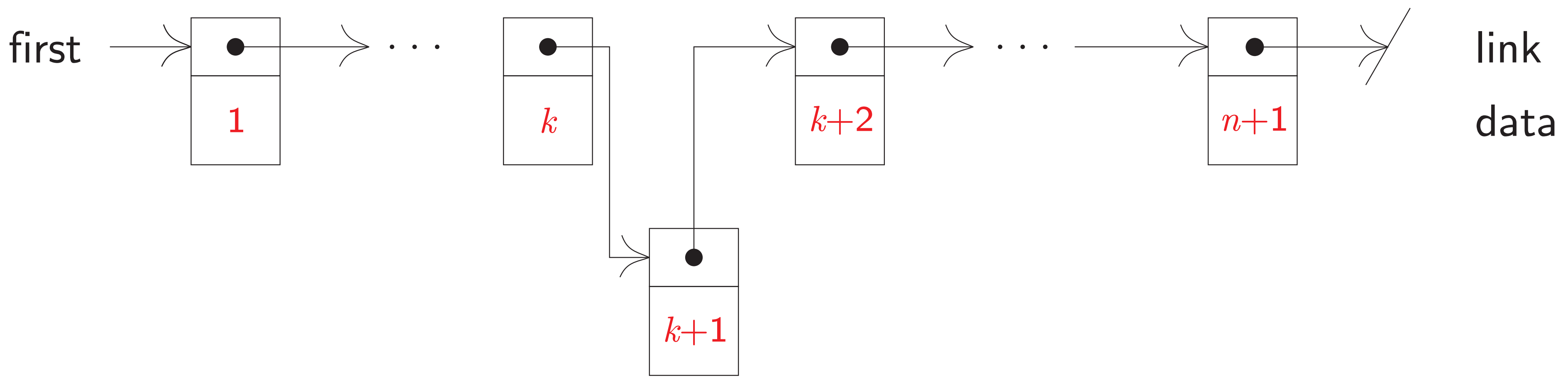


(b) After

If k≠0



(a) Before

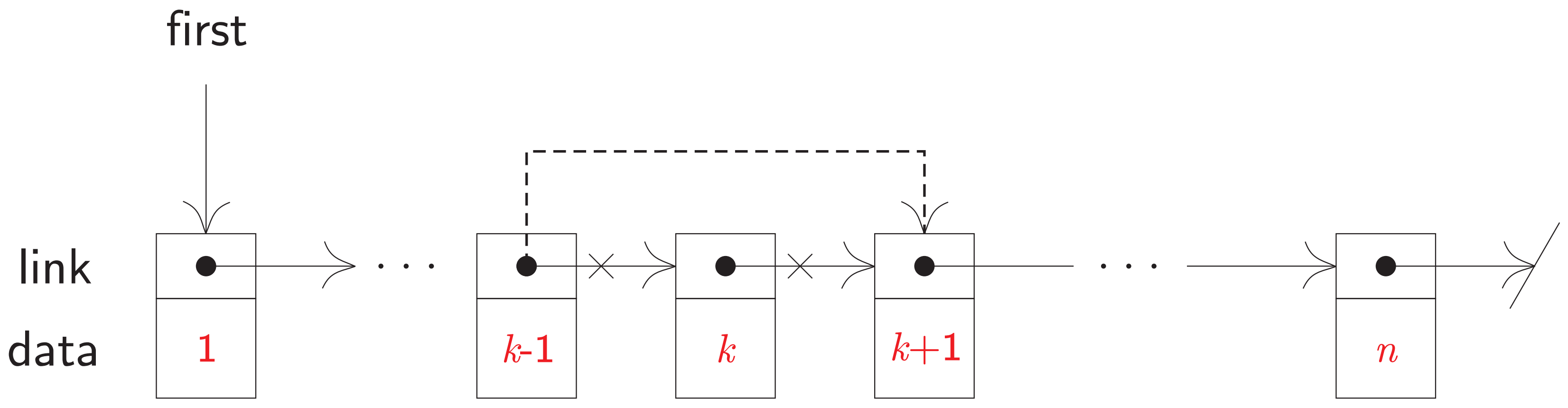


(b) After

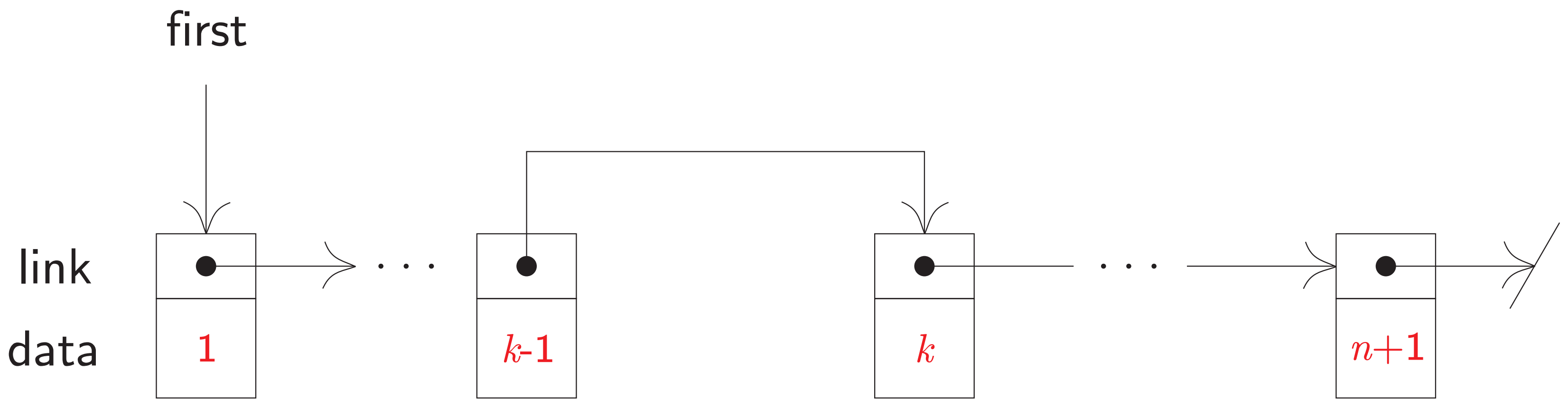
add

```c
void add( List * x, int index, int e ){
   int i;
   if( index < 0 || index > x->n ){
      fprintf( stderr, "Error␣:␣invalid␣index\n" );
      return;
   }
   Node * y = malloc( sizeof( Node ) );
   y->item = e;
   if( index == 0 ){
      y->next = x->firstNode;
      x->firstNode = y;
   }
   else{
      Node * p = x->firstNode;
      for( i = 0; i < index - 1; i++ )
         p = p->next;
      y->next = p->next;
      p->next = y;
   }
   x->n++;
   return;
}
```

Deleting the kth element



(a) Before



(b) After

delete

```
void delete( List * x, int index ){
   int i;
   if( index < 0 || index >= x->n ){
      fprintf( stderr, "Error␣:␣invalid␣index\n" );
      return;
   }
   if( empty( x ) ){
      fprintf( stderr, "Error␣:␣list␣is␣empty\n" );
      return;
   }
   Node * y;
   if( index == 0 ){
      y = x->firstNode;
      x->firstNode = x->firstNode->next;
   }
   else{
      Node * p = x->firstNode;
      for( i = 0; i < index - 1; i++ )
         p = p->next;
      y = p->next;
      p->next = p->next->next;
   }
   free( y );
   x->n--;
   return;
}
```

get, search

```
int get( List * x, int index ){
   int i;
   if( index < 0 || index >= x->n ){
      fprintf( stderr, "Error␣:␣invalid␣index\n" );
      return;
   }
   Node * p = x->firstNode;
   for( i = 0; i < index; i++ )
      p = p->next;
   return p->item;
}

int search( List * x, int e ){
   int i;
   Node * p = x->firstNode;
   for( i = 0; i < x->n; i++ ){
      if( p->item == e )
         return i;
      p = p->next;
   }
   return -1;
}
```

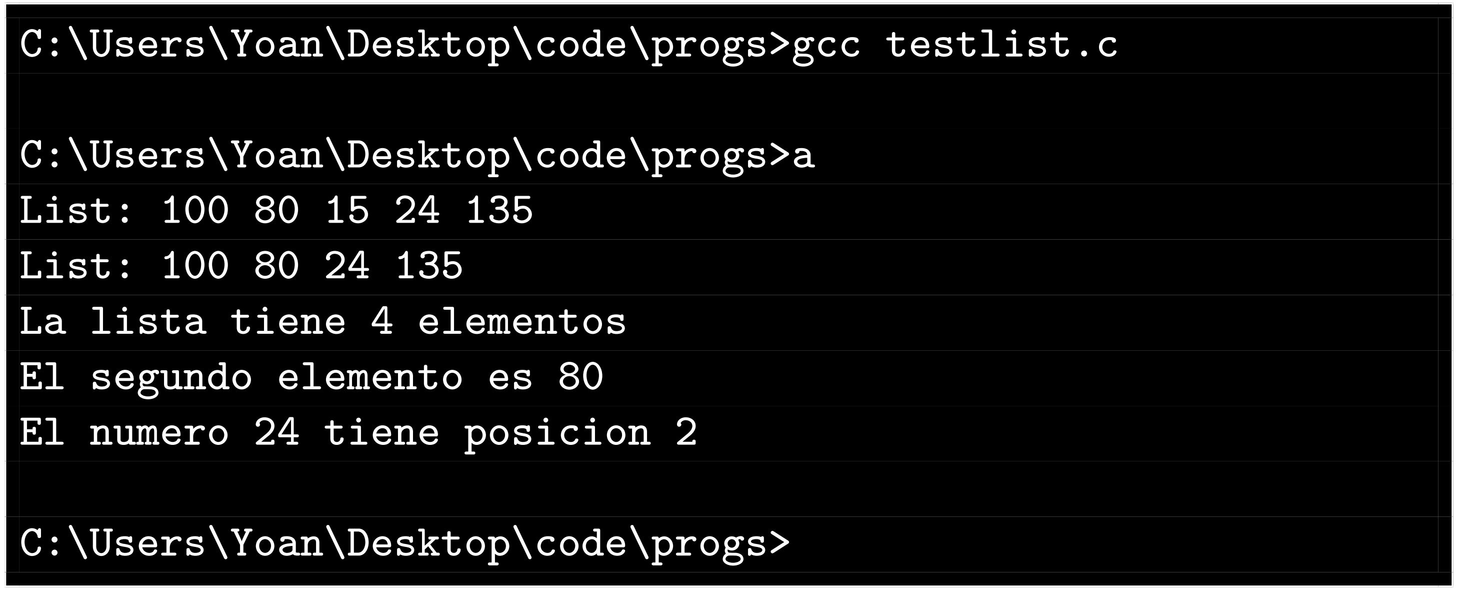
display, size, empty

```
void display( List * x ){
   int i;
   printf( "List:␣" );
   Node * y = x->firstNode;
   for( i = 0; i < x->n; i++ ){
      printf( "%d␣", y->item );
      y = y->next;
   }
   printf( "\n" );
   return;
}


int size( List * x ){
   return x->n;
}


int empty( List * x ){
   return x->n == 0;
}
```

testlist.c

```c
#include <stdio.h>
#include "linkedlist.h"

int main(){
   List maria = createList( );
   maria.add( &maria, 0, 100 );
   maria.add( &maria, 1, 80 );
   maria.add( &maria, 2, 15 );
   maria.add( &maria, 3, 24 );
   maria.add( &maria, 4, 135 );
   maria.display( &maria );
   maria.delete( &maria, 2 );
   maria.display( &maria );
   printf( "La␣lista␣tiene␣%d␣elementos\n", maria.size( &maria ) );
   printf( "El␣segundo␣elemento␣es␣%d\n", maria.get( &maria, 1 ) );
   printf( "El␣numero␣24␣tiene␣posicion␣%d\n", maria.search( &maria, 24 ) );
   return 0;
}
```

Compilation and Running



```
C:\Users\Yoan\Desktop\code\progs>gcc testlist.c

C:\Users\Yoan\Desktop\code\progs>a
List: 100 80 15 24 135
List: 100 80 24 135
La lista tiene 4 elementos
El segundo elemento es 80
El numero 24 tiene posicion 2

C:\Users\Yoan\Desktop\code\progs>
```

Complexity of Operations

add	:	$O(n)$
delete	:	$O(n))$
get	:	$O(n)$
search	:	$O(n)$
display	:	$\Theta(n)$
size	:	$\Theta(1)$
empty	:	$\Theta(1)$
createList	:	$\Theta(1)$

Array-based List Vs Linked List

- Arrays are easy to use, but they have a fixed length
- Can you predict the maximum number of element in the list?
- Will an array waste storage?
- Linked lists do not have a fixed length
- An array-based implementation does not waste space storing pointers to the next item
- You can access any array item directly with equal access time, whereas you must traverse a linked list to access the ith element

- Inserting/Deleting elements in an array based implementation requires the movement of other elements in the array
- Inseting/Deleting elements in a linked implementation does not require the movement of other elements in the array, however the appropriate position has to be found

$$\boxed{\text{If most of the activity is on inserting/deleting elements, there is little difference in performance (theoretically)}}$$

$$\boxed{\text{If most of the activity is on querying the list by their position, then the array-based implementation will have better performance}}$$