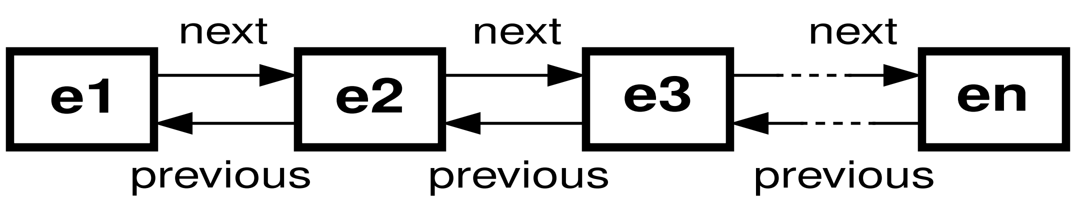
Lists, Stacks and Queues

Objectives:

- to define the list, stack and queue ADT's
- to present different data structures for implementing those ADT's
- to analyze and compare the data structures presented

Lists

- a list is a **finite, ordered sequence** of data items



head **tail**

n - length of the list
n = 0 - empty list

- **ordered list:** $e_i <= e_{i+1}$ for all i between 1 and n-1 (“<=” is an ordering relation over the set of possible elements)

The List ADT

Type: a collection of lists

Operations:

- length

$$\text{length}(e_1,e_2,\ldots,e_n) = n$$

- return element

$$\text{return}(k,(e_1,e_2,\ldots,e_k,\ldots,e_n))=e_k$$

- insert element

$$\text{insert}(x,k,(e_1,e_2,\ldots,e_{k-1},e_k,\ldots,e_n))=(e_1,e_2,\ldots,e_{k-1},x,e_k,\ldots,e_n)$$

- remove element

$$\text{remove}(k,(e_1,e_2,\ldots,e_k,\ldots,e_n))=(e_1,e_2,\ldots,e_{k-1},e_{k+1},\ldots,e_n)$$

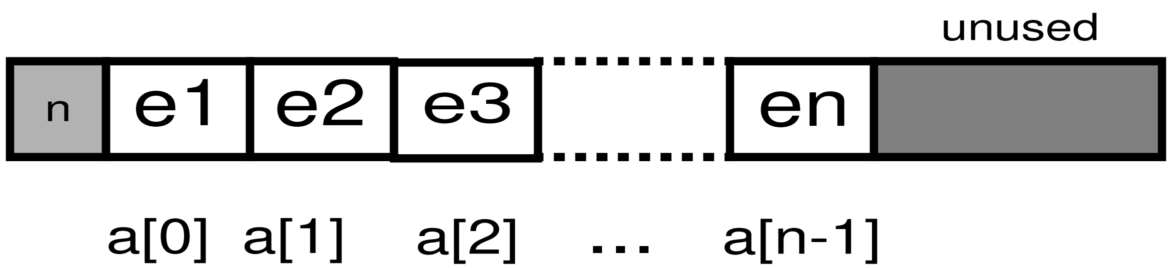
- find element

$$\text{find}(x,(e_1,e_2,\ldots,e_{k-1},x,e_{k+1},\ldots,e_n))=k \text{ (k - first occurrence)}$$

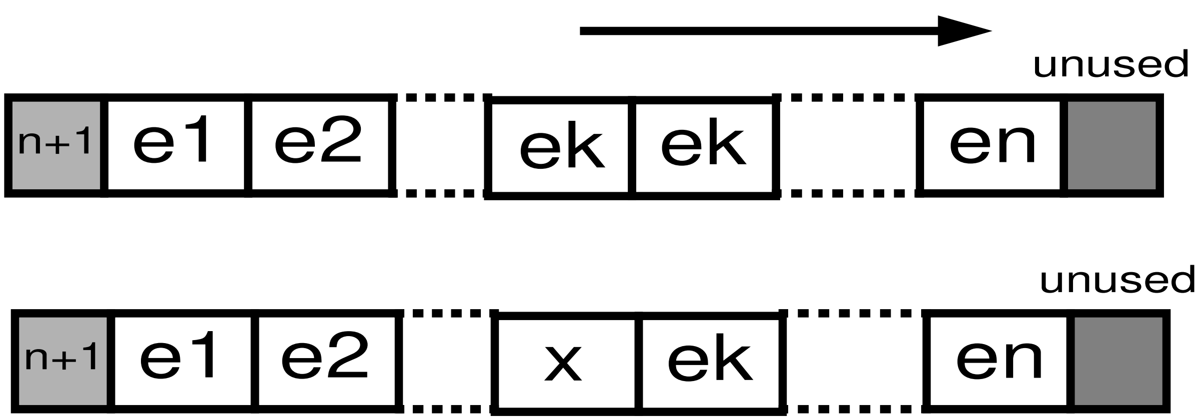
$$\text{find}(x,(e_1,e_2,\ldots,e_n))=0 \text{ (x - not in the list)}$$

Array-Based Implementation

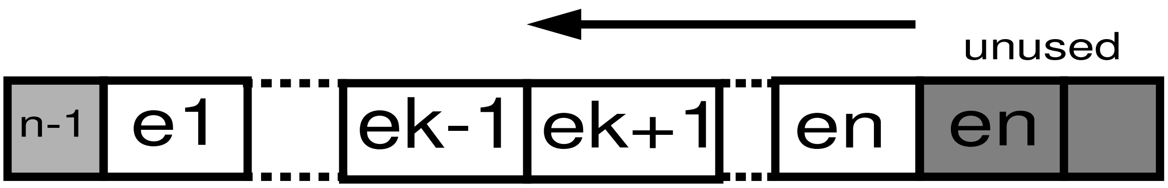
- list is stored in an array



- **length:** simply returns n
- **return element:** returns a[k-1]
- **insert element:** needs to shift n-k elements one to the right



- **remove element:** needs to shift n-k elements one to the left



- **find element:** visit the elements by incrementing index

```
i := 0
while (x <> a(i))
    i := i +1
return i;
```

C++ class definition

```
    #define Element int

class List{
private:
  int max_size;
  int actual_size;
  Element* list_array;
public:
  List(const int);
  ~List();
  length();
  Element& return_nth(const int);
  void insert(const Element&, const int);
  void remove(const int);
  int find(const Element&);
};
```

C++ class definition (continued)

```
List::List(const int size){
  max_size = size;
  actual_size = 0;
  list_array = new Element(size);
  return;
}

List::~List(){
  delete ()list_array;
  return;
}

int List::length(){
  return actual_size;
}

Element& List::return_nth(const int k){
  return list_array(k);
}
```

C++ class definition (continued)

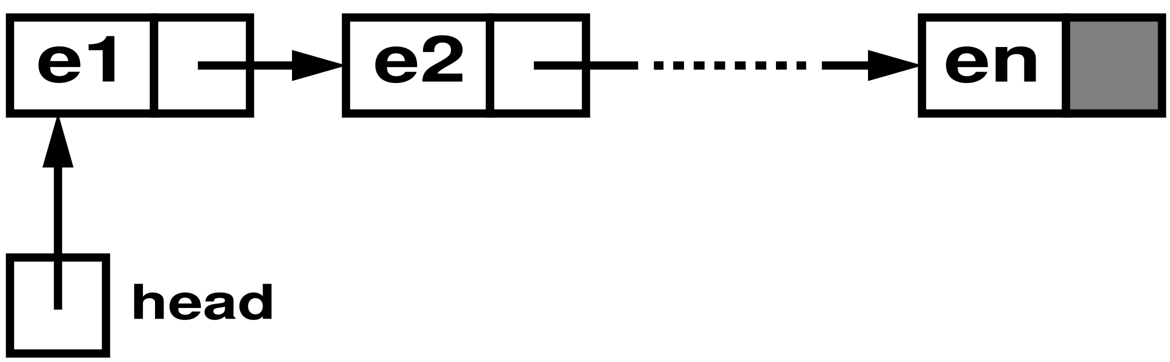
```
void List::insert(const Element& e,
                                          const int k){
  register int k_new;

  if (actual_size == max_size)
    return;
  else{
    actual_size++;
    k_new = (k < 0) ? 0 : k;
    for (int i = actual_size-1 ; i > k  ; i--)
      list_array(i) = list_array(i-1);
    list_array(k) = e;
    return;
  }
}
```

C++ class definition (continued)

```
void List::remove(const int k){
  if (actual_size == 0)
    return;
  else{
    actual_size--;
    for (int i = k ; i < actual_size - 1 ; i++)
      list_array[i] = list_array[i+1];
    return;
    }
}
int List::find(const Element& e){
  register int i;
  for (i = 0 ; (i < actual_size) &&
                    (list_array[i] != e) ; i++);
  if (list_array[i] == e)
    return i;
  else
    return -1;
}
```

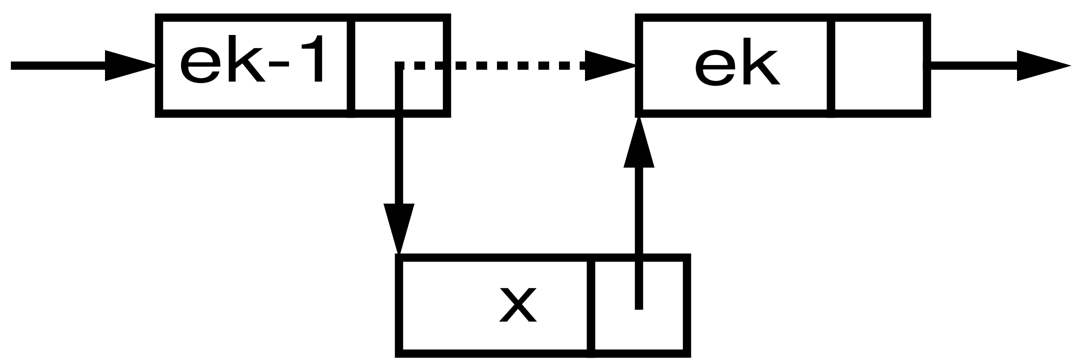
Linked Lists

- each element is stored in a separate structure (node); the ordering relation is implemented through pointers

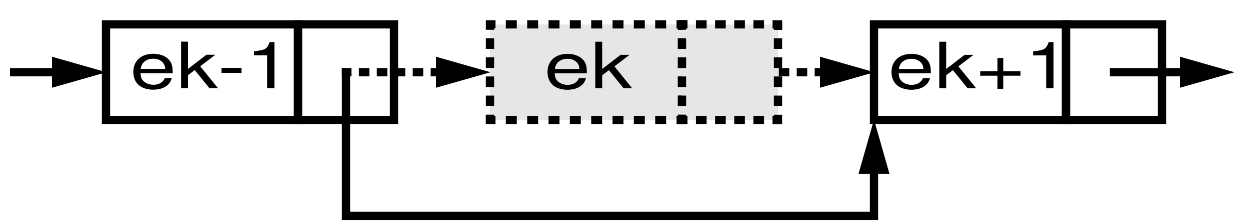


- **length:** can be stored in a separate field, otherwise needs to be counted
- **return element:** element k is accessed by following the links starting from the head

- **insert element:** after accessing the kth element, a new node needs to be created and the links changed



- **remove element:** the node corresponding to the element needs to be freed and the links changed



- **find element:** visit the elements by following the links, starting from the head (counting in parallel)

```
current := head
i := 1
while (current.value <> x)
    current := current.next
    i := i + 1
end (while)
return i;
```

C++ class definition

```
class Node{
public:
  Element value;
  Node *next;
 Node(const Element& e, Node* l = NULL)
         {value = e; next = l;}
  ~Node() {};
};
```

C++ class definition (continued)

```
class List{
private:
  int actual_size;
  Node* head;
public:
  List();
  ~List();
  length();
  Element& return_nth(const int);
  void insert(const Element&, const int);
  void remove(const int);
  int find(const Element&);
};
```

C++ class definition (continued)

```
List::List(){
  actual_size = 0;
  head = NULL;
  return;
}

List::~List(){
  for (Node *current = head ; head !=
NULL ; delete current)
    head = head -> next;
  return;
}
```

C++ class definition (continued)

```
int List::length(){
 return actual_size;
}

Element& List::return_nth(const int k){
 Node* current = head;
 for (int i = 0 ; i != k ; i++)
   current = current -> next;
 return current-> value;
}
```

C++ class definition (continued)

```
void List::insert(const Element& e,
                                              const int k){
  Node* current = head;

  actual_size++;
  if (k <= 0){
    head = new Node(e,head);
    return;
  }
  else{
    for (int i = 1 ; (i < k) &&
                (current->next != NULL) ; i++)
      current = current->next;
    current->next =
                new Node(e,current->next);
    return;
  }
}
```

```
void List::remove(const int k){
  Node* current = head;
  Node* temp;
   if ((k < 0) || (actual_size <= k))
    return;
  else{
    actual_size--;
    if (k == 0){
      temp = head;
      head = head->next;
    }
    else{
      for (int i = 1 ; i < k ; i++)
      current = current->next;
      temp = current->next;
      current->next = (current->next)->next;
    }
  }
  delete temp;
  return;
}
```

C++ class definition (continued)

```
int List::find(const Element& e){
  Node* current = head;
  register int i;

  for (i = 0 ;(current->value != e) &&
              (current->next!= NULL) ; i++)
    current = current->next;
  if (current->value == e)
    return i;
  else
    return -1;
}
```

Comparison of list implementations

- the size of an array-based list has to be predetermined
- a linked list only needs space for the elements actually on it
- array-based lists don’t require extra space for each element
- linked lists required that a pointer be added to each element

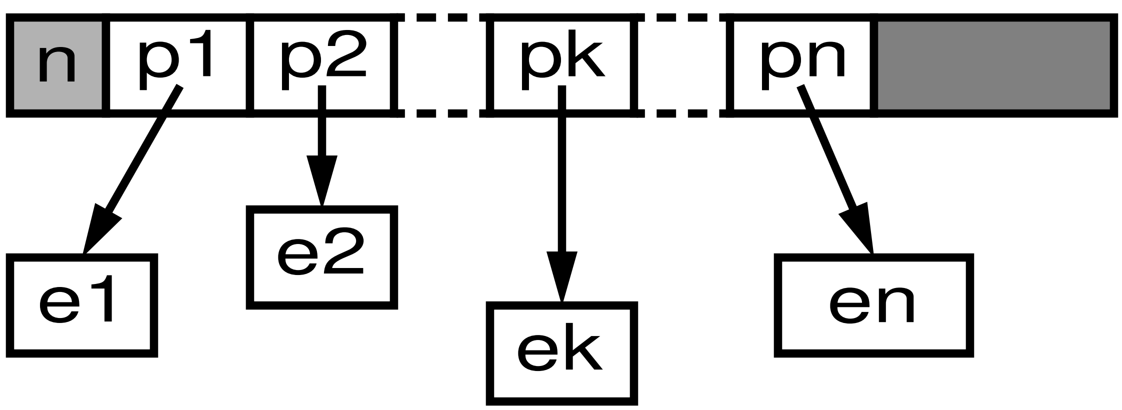
From the point of view of space required array-based lists are better when the size of the list is approximately known in advance

- elements in an array-based list can be accessed fast by indexing
- accessing an element in a linked list requires visiting the elements preceding it
- inserting and removing an element for an array-based list requires shifting part of the array
- inserting and removing an element for a linked list requires a fixed number of operations

Linked lists are preferred to array- based lists from the point of view of time required

Implementing elements

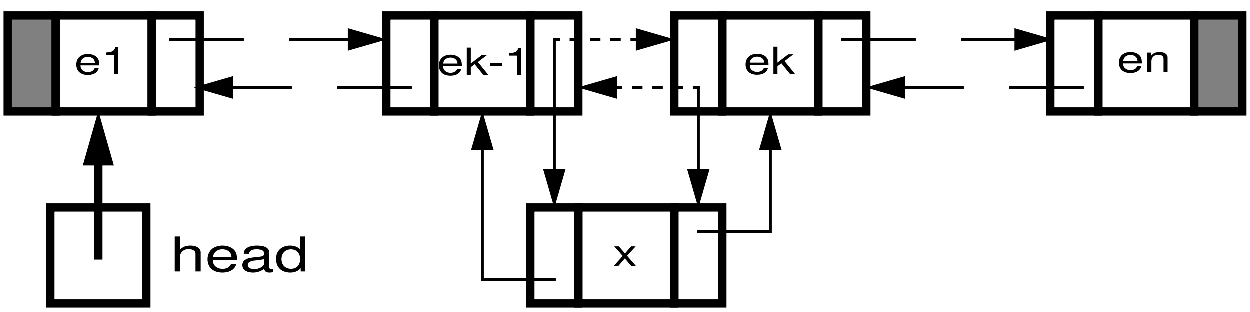
- for lists with large elements (such as strings or structures) storing pointers to the elements may be useful

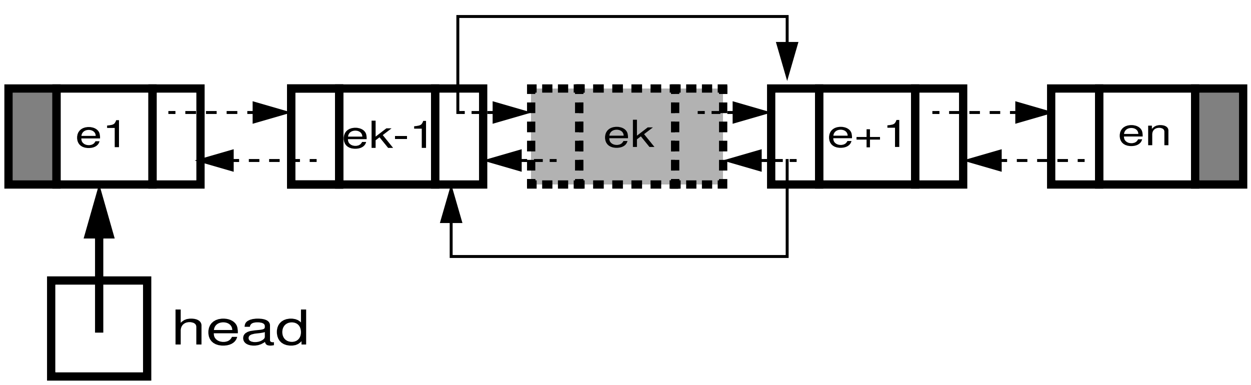


- this is advantageous when a certain element is repeated in the list (e.g. strings)

Doubly linked lists

- linked lists only allow the direct access from a node to the next node
- the insertion and deletion of elements in a linked list are complicated





Stacks

A stack is a list in which elements may be inserted or removed from only one end.

- operations on stacks have special names:

  push for insertion

  top for access

  pop for access & remove

- stacks can be implemented both as arrays and as linked lists

Queues

A queue is a list in which an element may only be inserted at the end and may only be accessed and removed at the head

- operations on queues have special names:

  enqueue for inserting

  dequeue for access & remove

- queues can be implemented both as arrays and as lists
- the array-based queue implementation raises special problems (drifting queue)