Machine Translated by Google

Chapter 4

Listen

Machine Translated by Google

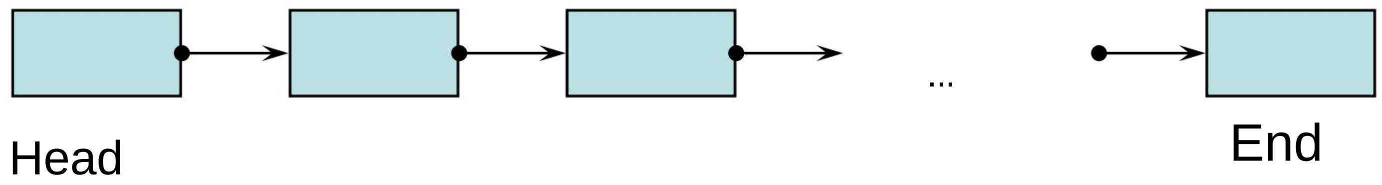
4.1 Definition Listen

A (linear) list is a data structure for managing an **arbitrarily large number** of elements of a **uniform type.**

Access to the individual elements of a (simple) list is only possible from **Head** off possible.

The **end** of the list is also called the **tail .**

The elements are arranged in a **sequence** that can (usually) be derived from the **order of entries** (disordered).



Machine Translated by Google

abstraction

Data structures represent an abstraction of a

conceptual model

Concept of **Abstract Data Type (ADT)**

Says nothing about the physical realization on the computer

Various realizations conceivable!

Realization often depends on the problem, programming environment, objectives, ...

Possible conceptual model “list”

“String of pearls”, beads are threaded at one end

Machine Translated by Google

List

The definition:

- A list L is an ordered set of elements L = (x1 , x2 , …, xn)

- The length of a list is given by |L| = |(x1 , x2 , …, xn)| = n

- An empty list has length 0.

- The ith element of a list L is denoted by L[i], therefore 1 ÿ i ÿ |L|

Machine Translated by Google

Methods on ADT list

Insert

Add: Insert element at the head

access

FirstElement: Determine head element

Delete

RemoveFirst: Remove head element

Generate

Constructor: Create new list

Length determination

Length: Determine the number of elements

Inclusion test

Member: Test whether element is included

Others

Operations conceivable

Machine Translated by Google

Great lists

universität wien

Declaration C++, Class

```
typedef … ItemType;
…
class List {
public:
    List(); // Constructor void
    Add(itemType a);
    ItemType FirstElement(); void
    RemoveFirst();
    int Length(); int
    Member(itemType a);
}
```

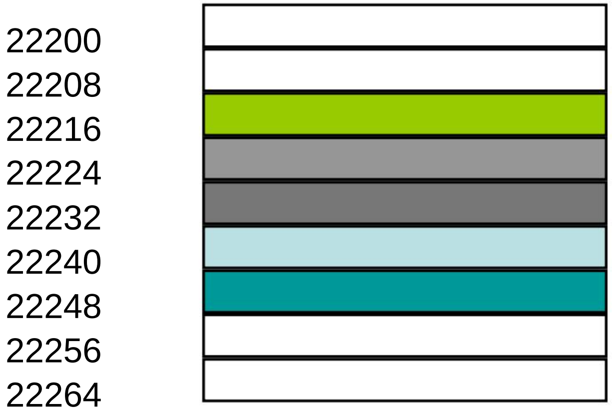
for use in classes
Def., better approach
with C++ templates

Machine Translated by Google

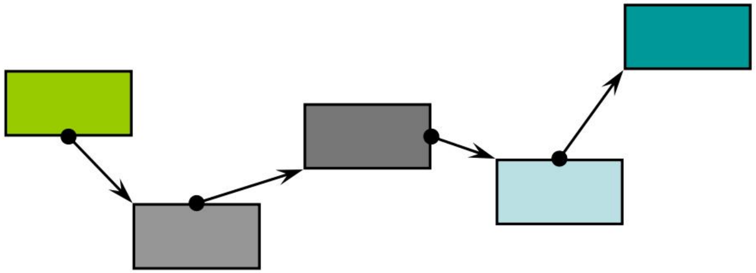
4.2 Implementation of lists

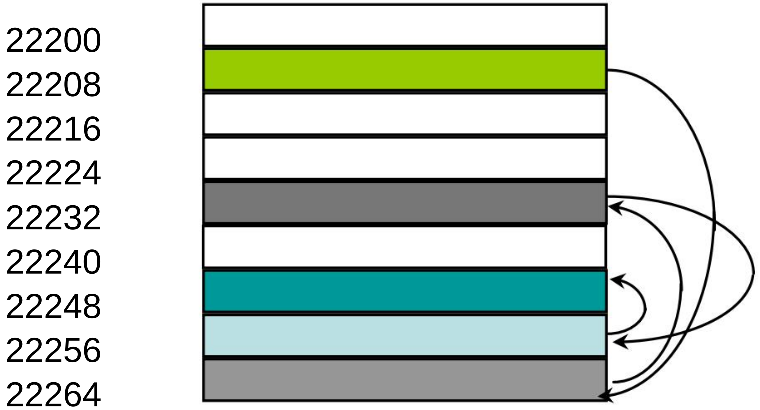
Storage types

Contiguous memory



Scattered (Linked) memory





Machine Translated by Google

Memory Typen

Contiguous memory

Physically **contiguous** storage area, extremely rigid as the final size is fixed when created.

Management via the **system.**

Data structures based on contiguous memory can only do one accommodate **a limited** number of items.

Scattered (Linked) memory

Physically distributed storage area, very **flexible** because of the extent can be **dynamically** adjusted.

Management (usually) via the **application program.**

Data structures can be **of any size .**

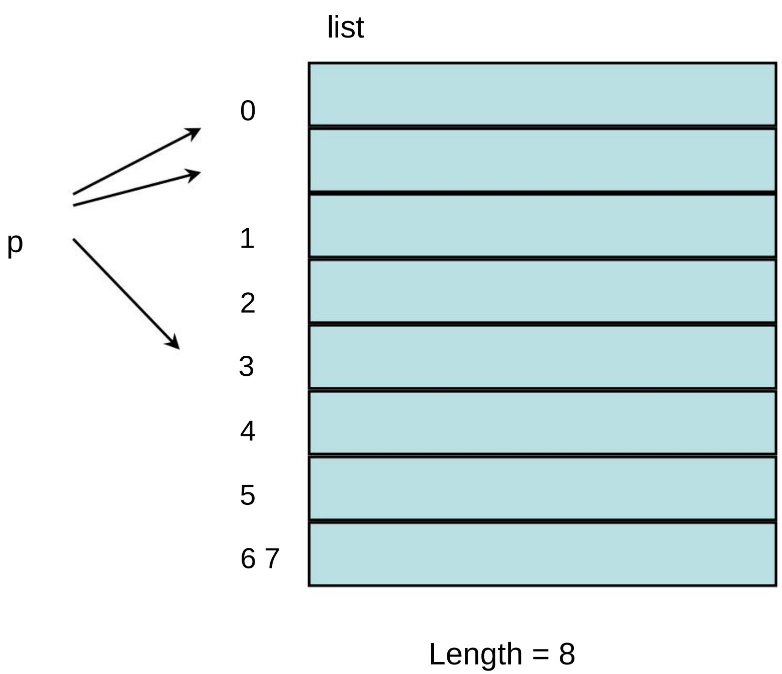
Machine Translated by Google

4.2.1 List - static - structure

Statische Implementation - contiguous memory

Store the elements in a field of limited length

```
typedef int ItemType; class
List {
private:
    ItemType list[8];
        // data structure
   int p; //
        next free position
    ...
}
```



Machine Translated by Google

List - static - Create - Destroy - Insert

Generate,

```
List::List() {p = 0;}
```

Destroy

```
List::~List() {p = 0;}
```

Insert

```
void List::Add(ItemType a) { if(p < 8) {

        list[p] = a;
        p++;

    } else cout << "Error-add\n";
}
```

Machine Translated by Google

List - static - access - delete

access

```
ItemType List::FirstElement() { if(p > 0)
    return list[p-1]; else cout << "Error-
    first\n";
}
```

Delete

```
void List::RemoveFirst() {
    if(p > 0) p--; else
    cout << "Error-remove\n";
}
```

Machine Translated by Google

List - static - Length - inclusion test

Long,

```
int List::Length() {
    return p;
}
```

Inclusion test

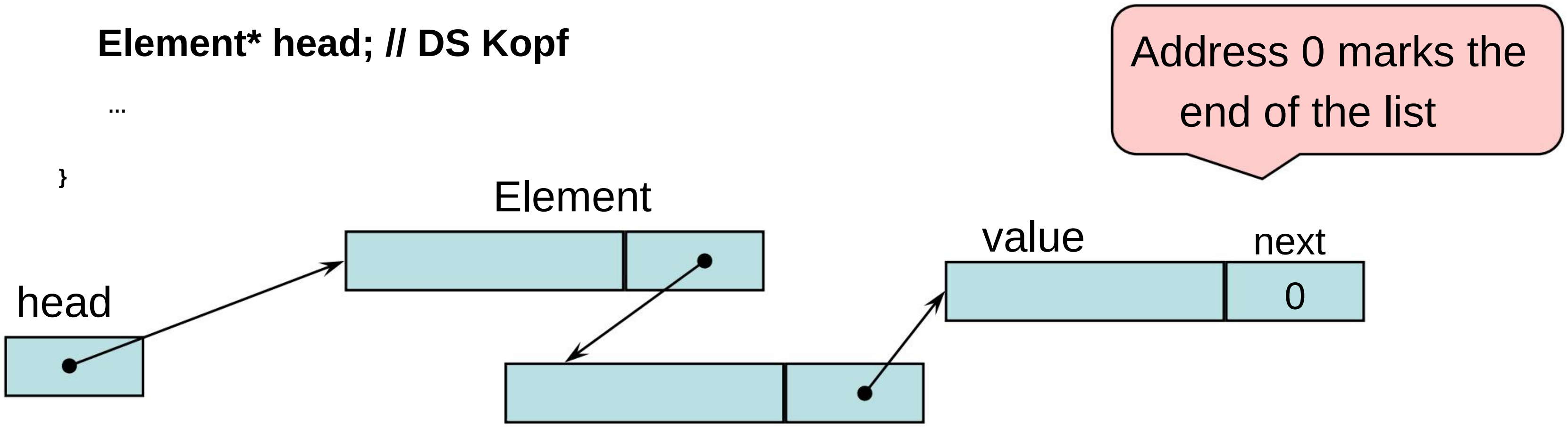
```
int List::Member(ItemType a) {
    int i = 0;
    while(i < p && list[i] != a) i++; if(i < p) return 1;
    else return 0;


}
```

4.2.2 List - dynamic - structure (1)

Dynamische Implementation - linked memory

Dynamically expandable list of unlimited length

```
typedef int ItemType;
class List
{ public:
   class Element {                 // Element class
         ItemType value;
         Element* next;
   };
   Element* head; // DS Kopf
   ...
}
```



Machine Translated by Google

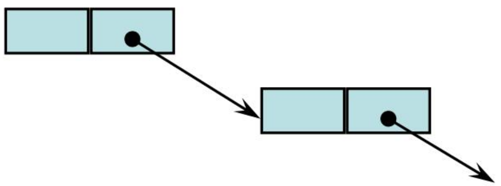
Recursive data structure

A data structure is called **recursive** or **circular** if it is in their definition itself references

Basic model for dynamically extendable data structures

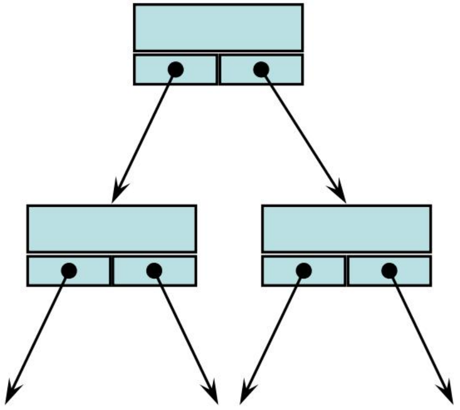
List

```
class Element{
    InfoType Info;
    Element* Next;
}
```



Baum

```
class Node {
    KeyType Key;
    Node* LeftChild;
    Node* RightChild;
}
```



Machine Translated by Google

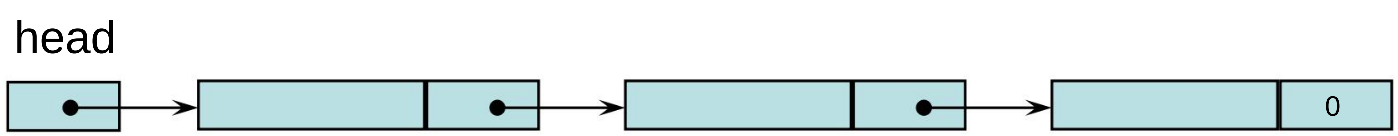
List - dynamic - insert

Insert

```
void List::Add(ItemType a) {
    Element* help; help
    = new Element; help->next
    = head; help->value = a;
    head = help;

}
```

Typically at the top of the list

Before inserting

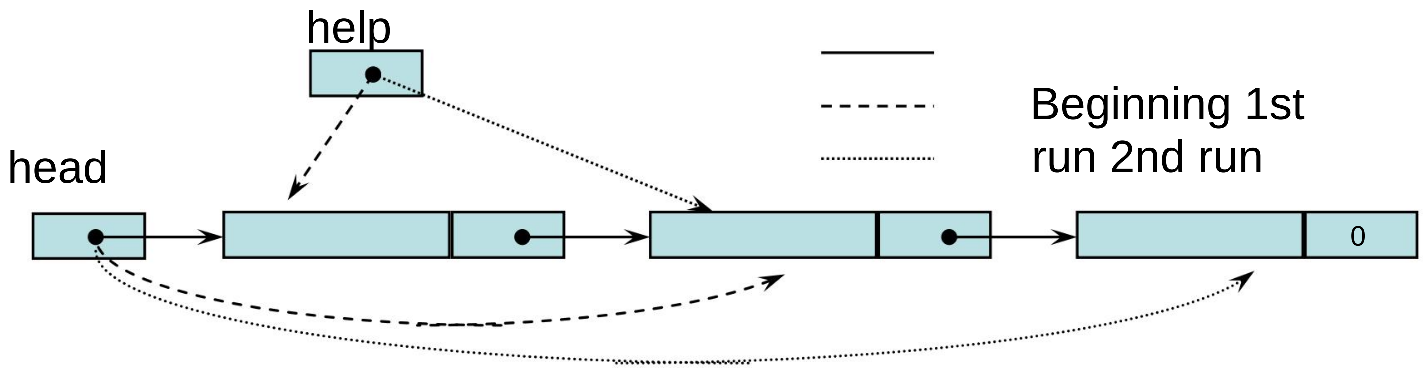


Machine Translated by Google

List - dynamic - Create - Destroy

Create, delete

```
List::List() { head = 0;}


List::~List() {
   Element* help;
   while(head != 0) {
     help = head;
     head = head->next;
     delete help;
   }
}
```



Machine Translated by Google

List - dynamic - access

access

```
ItemType List::FirstElement() {
    if(head != 0)
        return head->value;
    else
        cout << "Error-first\n";
}
```

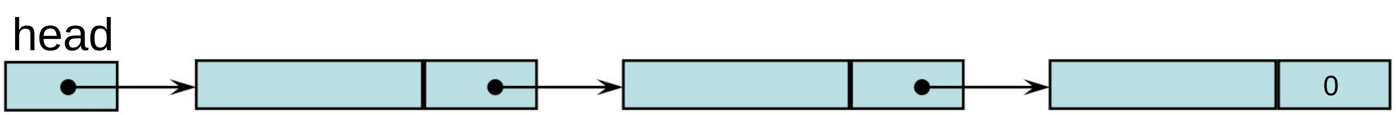
Machine Translated by Google

List - dynamic - delete

Delete

```
void List::RemoveFirst() {
    if(head != 0) {
        Element* help; help
        = head; head =
        head->next;
        delete help;
    } else cout << "Error-remove\n";
}
```

Before deleting



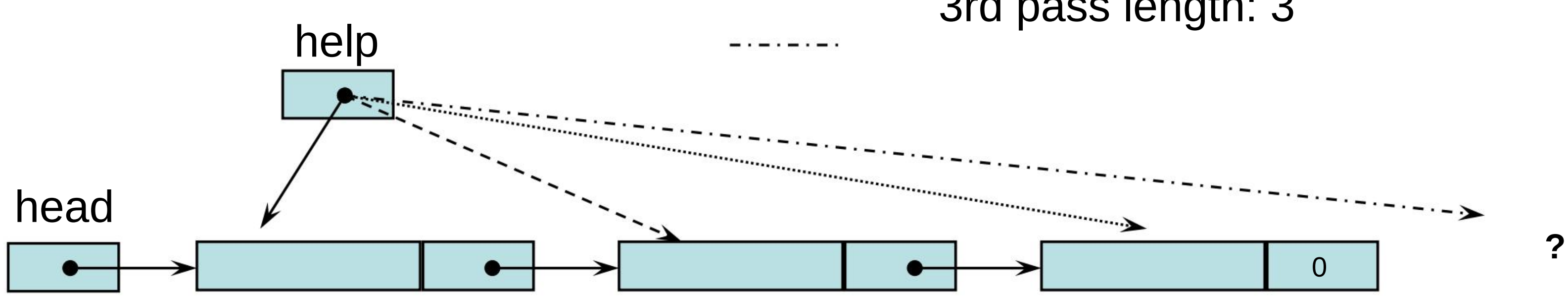
Machine Translated by Google

Length

```
int List::Length() {
    Element* help = head; int
    length = 0; while(help !
    = 0) {
      length++;
      help = help->next;
    }
    return length;
}
```

Checkpoint

Start, length: 0 1st pass
length: 1 2nd pass length: 2
3rd pass length: 3

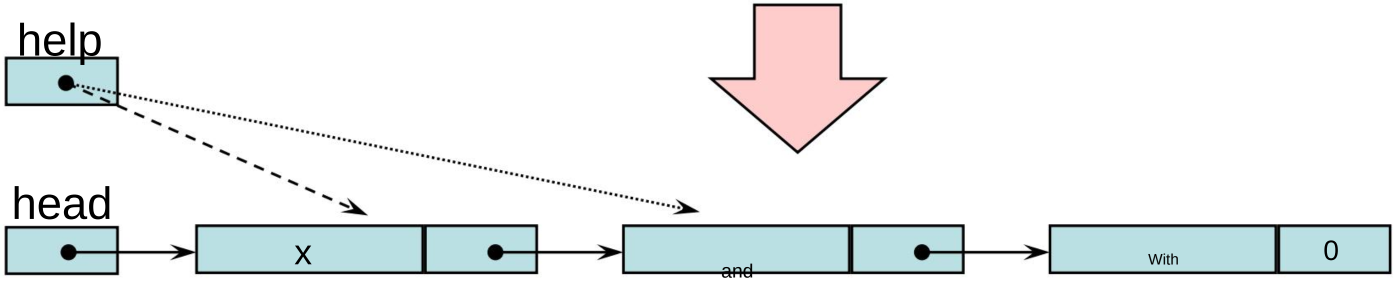


Machine Translated by Google

List - dynamic - inclusion test

Inclusion test

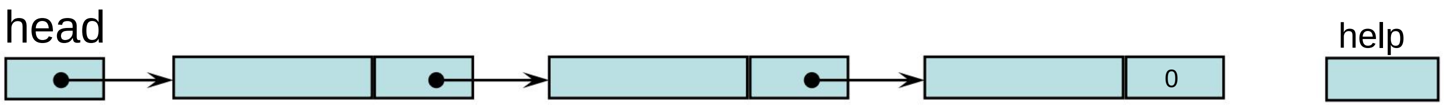
```
int List::Member(ItemType a) {
    Element* help = head;
    while(help != 0 && help->value != a)
        help = help->next;
    if(help != 0) return 1;
    else return 0;
}
```



Machine Translated by Google

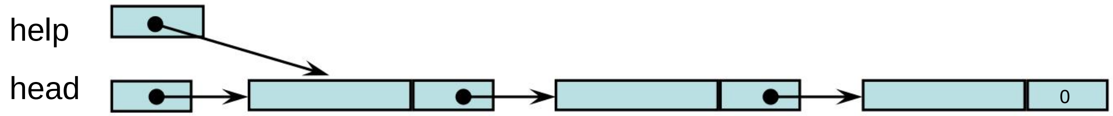
List - Ferries

Scheme for sequential processing of a list (iterative), i.e. “visiting all elements”



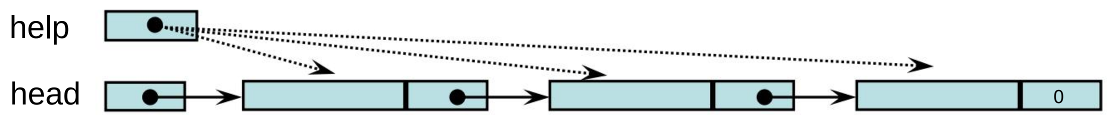
1. Initialize the auxiliary pointer

help = head;



2. Continue moving the auxiliary pointer (position)

help = help -> next;



3. Query for end of list (and search criterion)

while (help != 0 && …) { … }

Machine Translated by Google

4.3 Stack

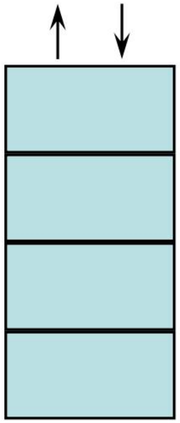
The **stack (basement memory)** is a special case of the list that contains the Elements managed according to the **LIFO** (last-in, first-out) principle

Idea of the stack: You can only access the top, most recently placed element (compare book stacks, piles of wood, ...)

Applications: pushdown machines, memory management, HP calculators (UPN), ...

The behavior of the stack can be described by its (quite simple) operations

- push: push element onto the stack
- top: Access the top element of the stack
- pop: Remove element from stack
- isEmpty: Test for empty stack

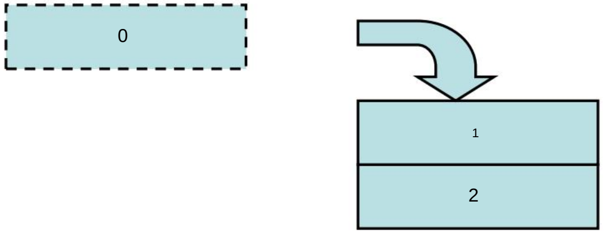


Machine Translated by Google

Methods on Stacks

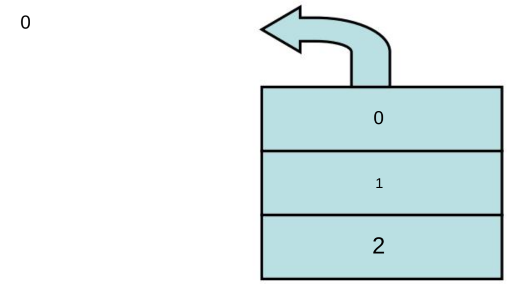
Methode ‘Push’

Element is placed on the stack (at the top position).



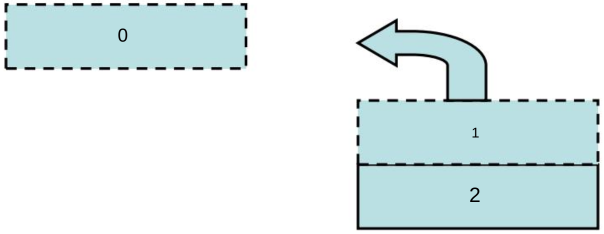
Method 'Top'

Returns the contents of the top element of the stack.



Method 'Pop'

Top element of the stack is removed.



Machine Translated by Google

Stack as a list

Stack is a special list, so the stack operations are expressed by list operations.

Push(S,a) ÿ Add(S,a)

Top(S) ÿ FirstElement(S)

Pop(S) ÿ RemoveFirst(S)

IsStackEmpty(S)

ÿ wenn Length(S) = 0 return true

otherwise false

Machine Translated by Google

4.4 Queue

The queue is a special case of the list that

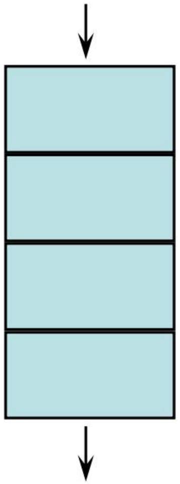
Elements managed according to the **FIFO** (first-in, first-out) principle

Idea: The elements are arranged one after the other, whereby elements can only be added to the end of the list and removed from the beginning of the list

Applications: Queues, buffer management, process management, metabolism,…

Simple operations

- Enqueue
  - Place element at the end of the queue
- Front
  - Access first element of the queue
- Dequeue
  - Remove first item from queue
- IsQueueEmpty
  - Test for empty queue



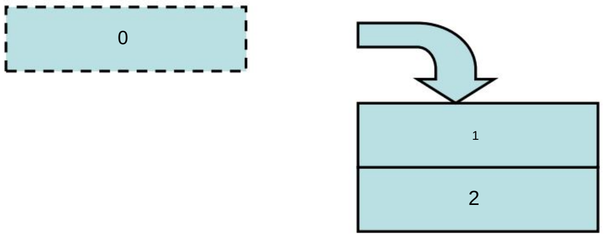
Machine Translated by Google

Methods on queue

Method 'Enqueue'

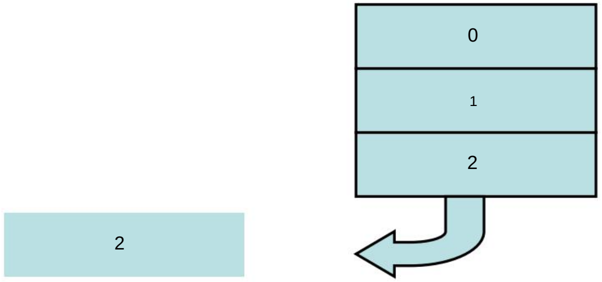
Element will be at the end of the

Queue placed (at last position)



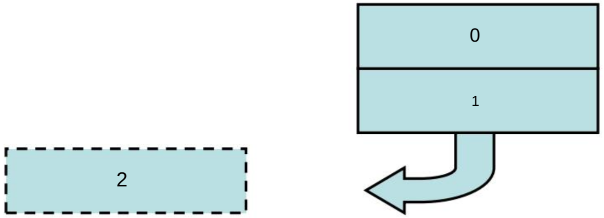
Method 'Front'

Delivers the content of the first element of the Queue



Method 'Dequeue'

First element of the queue is removed



Machine Translated by Google

Queue as a list

Queue is also a special list, so everyone should
Queue operations can also be expressed by list operations .

Enqueue(Q,a) ÿ Add(S,a)

Front(Q) ÿ ?

Accordingly (Q) ÿ ?

Not entirely trivial!

Possibility (inconvenient!)

Accessing the first queue element (last in the list) by iteratively removing all elements and simultaneously building a 'collapsed' auxiliary list. Then reverse the process.

Machine Translated by Google

Comparison of “our” data structures

General distinction between static and dynamic realization static R.:

contigous

memory, fields dynamic R.:

dynamic memory, dynamic objects

Data management

Insertion and deletion is supported

Data volume

static R.: limited, depending on the field size

dynamic R.: unlimited

depending on the size of the available storage space

rather simple models

Machine Translated by Google

Effort comparison of “our” lists Implementations

	List static	List dynamic
Storage space	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
destructor	O(1)	**O(n)**
Add	O(1)	O(1)
FirstElement	O(1)	O(1)
RemoveFirst	O(1)	O(1)
Length	O(1)	**O(n)**
Member	**O(n)**	**O(n)**

Attention: Actual effort O(n) in
Add and RemoveFirst hidden

Machine Translated by Google

4.5 Special Lists

Doubly Linked List

double linked list

Circular List

Circular verkettete Liste

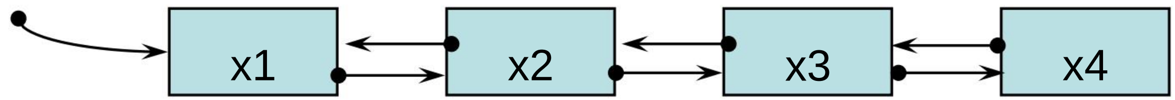
Ordered List

Ordered list

Double Ended List

Double headed list

Double linked list



Each element has 2 pointers, one pointing to the previous one and the other points to the subsequent element

Basic operations easy

```
class Node {
    KeyType Key;
    Node* Before;
    Node* Succ;
}
```

Machine Translated by Google

Circular List

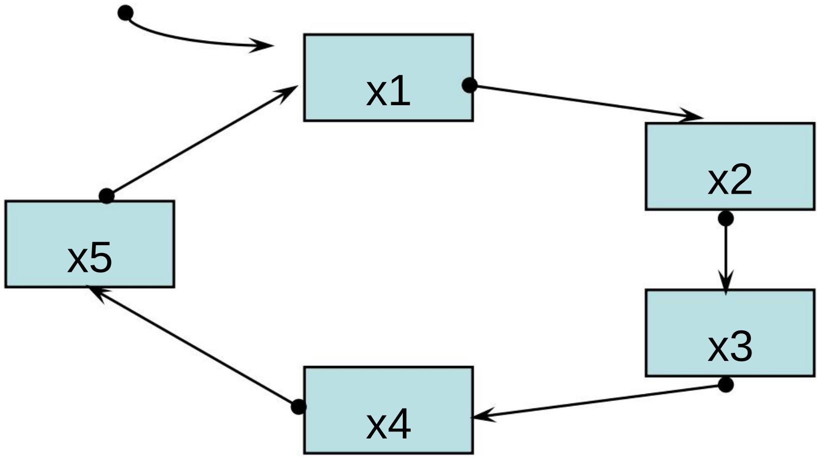
universität wien

Circular verkettete Liste

Pointer of the last element refers again to the first element Ring

Buffer Be

careful when entering and deleting the first element!

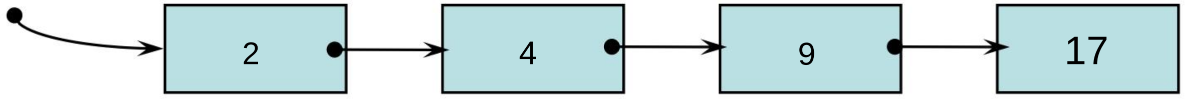


Machine Translated by Google

Ordered List

Ordered list

- Items are added to the list more specifically according to their value
  - Registered position
- Mostly ordered by size
- Enter in a specific place that has to be found first ÿ
  - traverse



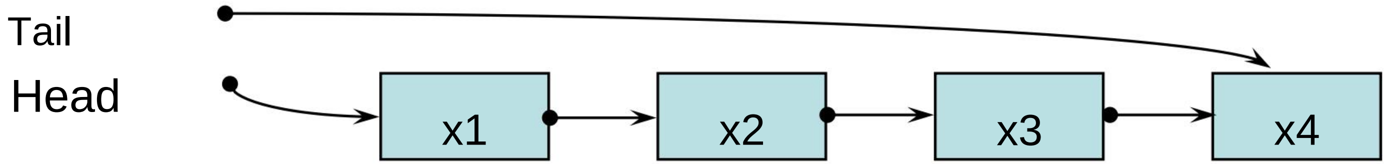
Machine Translated by Google

Double Ended List

List of 2 “heads”

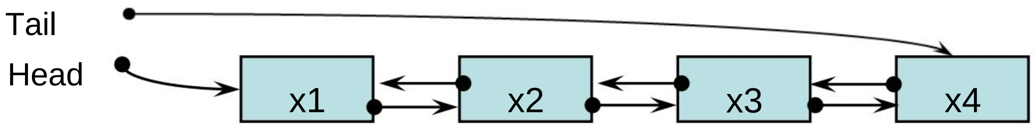
Each list has 2 pointers that point to the head and the end of the list

Makes it easier to insert at the top and bottom of the list



Can be combined with other list structures, e.g

Doubly Linked Double Ended List



Machine Translated by Google

What do we take with us?

Listen

- The operation
- storage
- Contigous - Scattered memory

Stack – Queue

Comparison

Special lists

- Doubly Linked List
- Circular List
- Ordered List
- Double Ended List