FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Algorithmen und Datenstrukturen

Kapitel 2: Elementare Datenstrukturen

Prof. Ingrid Scholl

FH Aachen - FB 5

scholl@fh-aachen.de

31.03.2020

Inhalt - Kapitel 2: Elementare Datenstrukturen

- Kapitel 2 - Elementare Datenstrukturen
  - Felder (Array)
  - Stapel und Warteschlangen
    - Stack mit Array-Implementierung
    - Interface, Implementierung, Client
    - Warteschlange als Array mit Ringpuffer
  - Prioritätenwarteschlange
  - Verkettete Listen
    - Einfach Verkettete Liste
    - Doppelt Verkettete Liste
    - Stapel und Warteschlangen mit Verketteten Listen

Dynamische Datenmengen

Definition (Dynamische Datenmengen)

Datenmengen, die in Algorithmen verwendet werden, können anwachsen, sich verringern oder über die Laufzeit auch verändern. Dies wird als dynamische Daten bezeichnet.

Definition (Dynamische Datenstrukturen)

Dynamische Datenmengen für Algorithmen und grundlegende Operationen, um auf die Daten zuzugreifen (Abfrage-Methoden) und modifizierende Methoden.

Grundlegende Operationen auf dynamischen Daten

1. $SEARCH(S, k)$
   Eine Abfrage, die zu einer Menge S und einem Schlüssel k eine Referenz x von einem Element aus der Menge S zurück gibt, so daß $key[x] = k$ oder NULL, wenn es den Schlüssel nicht in der Menge gibt.
2. $INSERT(S, x)$
   Eine modifizierende Operation, die die Menge S mit dem Element referenziert durch x erweitert. Alle Attribute von x wurden vorab initialisiert.
3. $DELETE(S, x)$
   Eine modifizierende Operation, die das durch x referenzierte Element aus der Menge S entfernt.

Weitere typische Operationen auf dynamischen Daten

4. *MINIMUM*(S)
   Eine Abfrage zur Ordnung von S, liefert einen Zeiger auf das Element von S mit dem kleinsten Schlüssel.
5. *MAXIMUM*(S)
   Eine Abfrage zur Ordnung von S, liefert analog den größten Schlüssel.
6. *SUCCESSOR*(S, x)
   Eine Abfrage zur Ordnung der Menge: Liefert zu einem gegebenen Element x einen Zeiger zum nächst größeren Element in S oder NULL, falls x das Maximum ist.
7. *PREDECESSOR*(S, x)
   Analog zur *SUCCESSOR*-Abfrage, liefert das Vorgängerelement oder Null, falls x das Minimum ist.

Felder (Array)

Definition (Array)

Ein Feld (Array) *a* besteht aus einer festen Anzahl von Datenobjekten gleichen Typs, die über einen Index *i* über einen direkten Zugriff selektiert werden können. Der benötigte Speicherplatz für das Array ist zusammenhängend.

Index	Datenelement
0	a_0
1	a_1
2	a_2
3	a_3
⋮	⋮
n-1	a_{n-1}

- Array-Deklaration:

```
int* a;
```

- Speicher allokieren:

```
a = new int[n];
```

- Array initialisieren:

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

Felder (Array) - C++ Beispiele (1)

```
// Allokiere und initialisiere 8 int-Objekte
int myArray[] = {12, 15, 17, 35, 39, 47, 78, 83};

// Feldlaenge bestimmen
const int myArrayLength =
(int) sizeof(myArray)/sizeof(myArray[0]);

// Zugriff auf 3. Element
int drittesElement = myArray[2];

// Zugriff mit Zeigern bzw. ueber Adressen
int *p1 = myArray;

int *p2 = &myArray[0]; // Zeiger auf 1. Feldelement = p1
int *p5 = p1 + 4;      // Zeiger auf das 5. Feldelement
int feldElement = *p5; // Inhalt vom Zeiger p5, hier 39
```

Felder (Array) - C++ Beispiele (3)

```
// Ausgabe von allen Feldelementen
int *p1 = myArray;
for (int i=0; i<myArrayLength; i++) {
   cout << "myArray-Element " << i << ": " << *p1;
   p1++;
}

// Zeiger in C++
int v = 5;
int *p = &v;  // p enthaelt Adresse von v
int v2 = *p;  // v2 enthaelt den Inhalt von der Adresse p,
              // hier v2 == 5
```

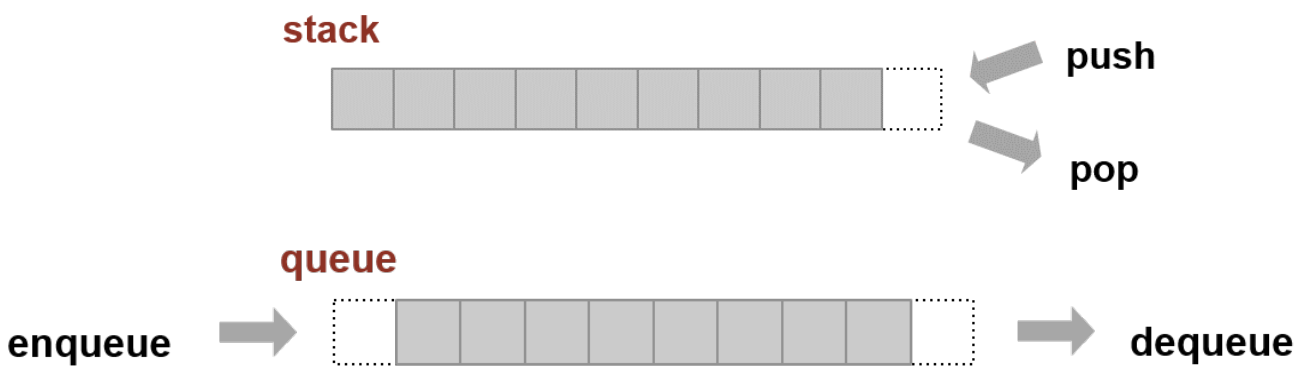
Felder (Array) - C++ Beispiele (3)

```
int a[4] = { 0, 1, 2, 3 };  int *ip;  int i;

for ( i=0; i<4; i++ ) cout << a[i] << "␣";
cout << endl;

ip = a;
for ( i=0; i<4; i++ ) cout << *ip++ << "␣";
cout << endl;

for ( i=0, ip=a; i<4; i++, ip++ ) cout << *ip << "␣";
cout << endl;

ip = &a[0];     /* entspricht  ip = a;  */
for ( i=0; i<4; i++ ) cout << ip[i] << "␣";
cout << endl;
```

Stapel und Warteschlangen (Stacks and Queues)

Stapel und Warteschlangen:

- Datenwerte: Sammlungen von Objekten gleichen Typs
- Operationen: Einfügen, Löschen, Iterieren, Test ob Datenstruktur leer
- Einfügeposition vorab bekannt
- Nächstes zu löschende Element vorbestimmt:
  Stack - LIFO-Prinzip "last in first out"
  Queue - FIFO-Prinzip "first in first out"

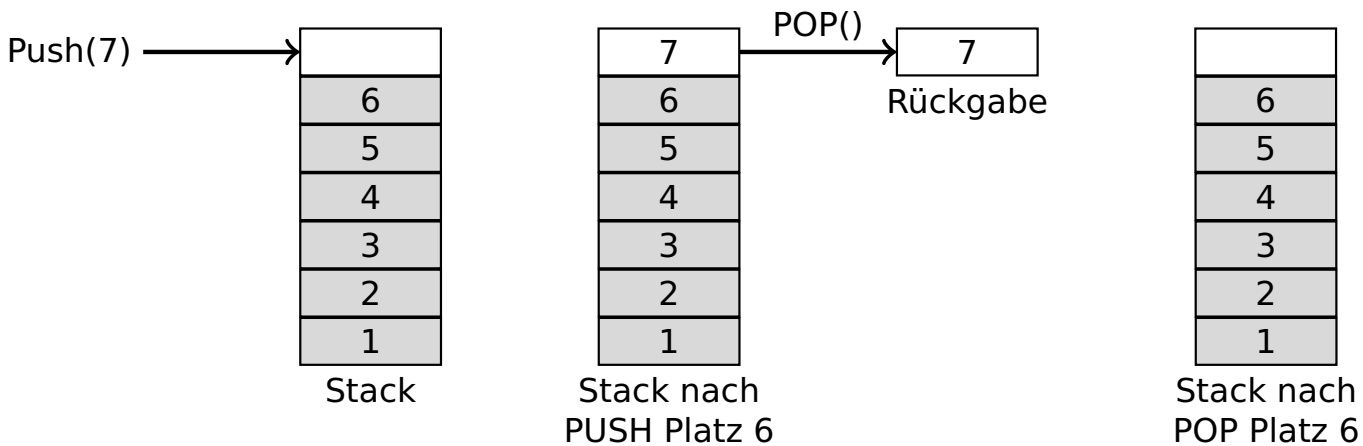


Stack

INSERT-Operation wird PUSH genannt.
DELETE-Operation wird POP genannt.

POP-Operation ist je nach Implementierung verschieden:

- liefert das letzte eingefügte Element zurück und löscht dieses gleichzeitig vom Stapel.
- liefert das letzte eingefügte Element zurück ohne dieses vom Stapel zu löschen (i.d.R. top()-Operation)



Stack mit Array-Implementierung

- DS mit max. n Daten ist hier ein Array S[0..n-1]
- Array hat einen Index-Zeiger top, der auf die letzte geschriebene Position verweist
- Stack besteht aus den Elementen S[0..top]
- S[0] ist das Element ganz unten im Stapel
- S[top] ist das Element ganz oben im Stapel
- Leerer Stapel bei top == -1,
  Underflow-Fehlermeldung beim Lese-Zugriff
- Voller Stapel bei top == n-1,
  Overflow-Fehlermeldung beim Schreib-Zugriff

Pseudocode Array-Implementierung Stack (1)

Algorithm 1: Stapel mit Array (1)

```
Function STACK_FULL(S)
    if top == n − 1 then
        return TRUE
    else
        return FALSE
Function STACK_EMPTY(S)
    if top == −1 then
        return TRUE
    else
        return FALSE
```

Pseudocode Array-Implementierung Stack (2)

Algorithm 2: Stapel mit Array (2)

```
Function PUSH(S, x)
    if STACK_FULL(S) then
        error "overflow"
    else
        top ← top + 1
        S[top] ← x

Function x = Pop(S)
    if STACK_EMPTY(S) then
        error "underflow"
    else
        top ← top − 1
        return S[top + 1]
```

Interface, Implementierung, Client

Definition

Interface: Definition der Datentypen und der grundlegenden Operationen (API - Application Interface)
Implementierung: Implementierung der Interface-Operationen
Client: Anwendung der Operationen aus dem Interface

Trenne Interface DS (Header-Datei) und Implementierung der DS (cpp-Datei) und Client (cpp-Datei, Programm, das die DS nutzt, i.d.R. hier die main).
Bsp: stack, queue, bag, priority queue, hash table, ...
Vorteile:

- Client benötigt keine Kenntnis von Implementierungs-Details der Datenstruktur
- Implementierung kennt keine Details vom Client
- Design: modular und wieder verwendbare Bibliotheken
- Performanz: verwendet effiziente Implementierungen

Bsp: Client

```
int main(int argc, char *argv[])
{
    Stack myStack(7);
    myStack.push(15);
    myStack.push(6);
    myStack.push(2);
    myStack.printAll();

    myStack.push(17);
    myStack.push(3);
    myStack.printAll();

    int x = myStack.pop();
    cout << "Entnommenes Element vom Stapel = " << x << endl;
    myStack.printAll();

    return 0;
}
```

Bsp: Interface

```
class Stack{
private:
    int* s; // Stapel als Array
    int  n; // Array-Groesse
    int  top; // Leseposition
public:
    Stack(int n);
    Stack();
    ~Stack();
    void push(int item);
    int pop();
    bool isEmpty();
    bool isFull();
    void printAll();
};
```

Bsp: Implementierung Interface (1)

```
Stack::Stack(){           // Konstruktor
   this->s = new int[10];
   this->n = 10;
   this->top = -1;
}

Stack::Stack(int n){      // Ueberladener Konstruktor
   this->s = new int[n];
   this->n = n;
   this->top = -1;
}

Stack::~Stack(){          // Destruktor
   delete [] this->s;
}
```

Bsp: Implementierung Interface (2)

```
void Stack::push(int item){
   if (isFull())
      cout << "overflow error: stack is full" << endl;
   else
   {  top = top + 1;
      s[top] = item; }
}

int Stack::pop(){
   if (isEmpty())
      cout << "underflow error: stack is empty" << endl;
   else
   {  top = top - 1;
      return s[top+1]; }
}
```

Bsp: Implementierung Interface (3)

```
bool Stack::isEmpty(){
    if (top == -1) return true;
    else return false;
}

bool Stack::isFull(){
    if (top == n-1) return true;
    else return false;
}

void Stack::printAll(){
    cout << "Stapel-Inhalt:␣";
    for (int i=0; i<=top; i++)
        cout << s[i] << "␣";
    cout << endl;
}
```

Bsp: Run Client

```
Stapel-Inhalt: 15 6 2
Insert Element 17
Insert Element 3
Stapel-Inhalt: 15 6 2 17 3
Entnommenes Element vom Stapel = 3
Stapel-Inhalt: 15 6 2 17
```

Arraydarstellung

Grafik der Stapelzustände

Bsp: Auswertung von Arithmetischen Ausdrücken

Nutze dazu 2 Stapel und wende die folgenden Regeln an:

- Operanden auf den Operandenstapel
- Operatoren auf den Operatorstapel
- Ignoriere linke Klammern
- Bei rechter Klammer: Entnehme Operator aus Operatorstapel, entnehme die entsprechende Anzahl an Operanden aus Operandenstapel, führe die Operation aus und lege das Ergebnis wieder auf den Operandenstapel zurück

Bsp: (1 + ((2+3)*(4*5)))

Bsp: Auswertung von Arithmetischen Ausdrücken

Bsp: (1 + ((2+3)*(4*5)))

Operandenstapel

Operatorstapel

Übungsaufgaben: Stapel

1. Stapel S sei als Array mit 6 Feldelementen implementiert. Zeigen Sie den Arrayinhalt, wenn nach der Reihe die folgenden Operationen ausgeführt werden: PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,8), POP(S).
2. Erläutere, wie man 2 Stapel mit 1 Array $A[0..n-1]$ implementieren kann, so daß weder ein Stack overflow noch die Gesamtanzahl der Elemente in beiden Stapeln n ergibt. PUSH- und POP-Operationen sollen in O(1) laufen.
3. Wandeln Sie arithmetische Ausdrücke mit einem Stapel von der Infix- zur Postfix-Notation um. Operatoren auf den Stapel, (überlesen,) Operator vom Stapel, Operanden direkt ausgeben. Bsp: $(a+(b*c)) \rightarrow abc*+$ $(a+((b*c)*(d-e))) \rightarrow abc*de-*+$

Definition Warteschlange

Definition (Queue)

Eine Warteschlange (Queue) ist eine lineare Liste, bei der Listenelemente nur an einem Ende (tail) eingefügt und nur am anderen Ende (head) entnommen werden.

In einer Queue wird immer dasjenige Element als nächstes gelöscht, das am längsten in der Queue vorhanden ist.
Prinzip: First-In, First-Out (FIFO)

Pseudocode Array-Impl. Queue (1)

Algorithm 3: Queue mit Ringpuffer-Array (1)

```
Function QUEUE_IsFull(Q)
    if head == tail then
        return TRUE
    else
        return FALSE
Function QUEUE_IsEmpty(Q)
    if (head + 1) mod n == tail then
        return TRUE
    else
        return FALSE
```

Pseudocode Array-Impl. Queue (2)

Algorithm 4: Queue mit Ringpuffer-Array (2)

```
Function QUEUE_Enqueue(Q, x)
    if !QUEUE_IsFull(Q) then
        Q[tail] ← x
        tail ← (tail + 1) mod n
        return TRUE
    else
        error "overflow"
Function x = QUEUE_Dequeue(Q)
    if !QUEUE_IsEmpty(Q) then
        head ← (head + 1) mod n
        return Q[head]
    else
        error "underflow"
```


FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

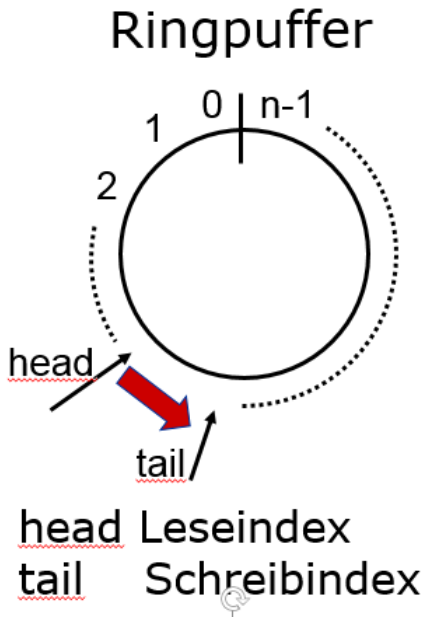
Definition API Warteschlange

```
class Queue {
private:
    ...
public:
    Queue();      // Konstruktur: erzeugt leere Warteschlange
    ~Queue();     // Destruktor: gibt Speicherplatz frei
    void enqueue(Item item); // Fuegt Element hinzu
    Item dequeue();  // Entfernt ein Element
    bool isEmpty();  // Abfrage, ob Queue leer
    int size();      // Aktuelle Anzahl Elemente in der Queue
};
```

In einer Queue wird immer dasjenige Element als nächstes gelöscht, das am längsten in der Queue vorhanden ist.
Prinzip: First-In, First-Out (FIFO)

Ringpuffer-Implementierung Warteschlange

```
class Queue {
private:
    Item* Q;
    int n;     // Array-Groesse
    int head; // Leseposition
    int tail; // Schreibposition
public:
    Queue(){
        Q = new Item[8];
        n = 8;     // max. Anzahl
        head = 7; // Leseindex
        tail = 0; // Schreibindex
    }
    ...
};
```



Übungsaufgaben: Warteschlange

1. Queue *Q* sei als Array mit 6 Feldelementen implementiert. Zeigen Sie den Arrayinhalt, wenn nach der Reihe die folgenden Operationen ausgeführt werden: ENQUEUE(Q,4), ENQUEUE(Q,1), ENQUEUE(Q,3), DEQUEUE(Q), ENQUEUE(Q,8), DEQUEUE(S).
2. Beachten Sie die Methoden ENQUEUE und DEQUEUE so, daß ein Underflow und ein Overflow einer Queue erkannt werden.
3. Programmieren Sie die Queue als Ringpuffer mit der Datenstruktur Array. Das Client-Programm (main) soll Overflow und Underflow entsprechend testen.

Prioritäts-Warteschlange (Priority Queue)

Definition (Priority Queue)

Eine Prioritätswarteschlange ist eine Datenstruktur von Elementen mit Prioritäten (Schlüsseln), die zwei grundsätzliche Operationen unterstützen:

- Einfügen eines neuen Elementes und
- Entfernen des Elementes mit der größten Priorität (Schlüssel).

Zwei Impl.-Strategien:

1. Sortiertes Einfügen nach Priorität mit $O(n)$ und Entnahme des Elementes mit höchster Priorität mit $O(1)$
2. Einfügen an beliebiger Position mit $O(1)$ und Entnahme des Elementes mit höchster Priorität mit $O(n)$

Datenstrukturen für die Priority Queue

Für die Priority Queue können verschiedene Implementierungen umgesetzt werden:

1. Array (effizienter bei unsortiertem Einfügen)
2. Verkettete Liste (effizienter beim sortierten Einfügen)
3. *später: PQ als Heap (Entnahme mit $O(\log n)$)*

Array-Datenelemente:

```
class PQItem{
public:
   int data;
   int prio;
};
```

Knoten der verketteten PQ:

```
class PQNode{
public:
   int data;
   int prio;
   PQNode *next;
};
```

PQ API mit Array

```
class PriorityQueue {
private:
   PQItem *pItem; // Zeiger auf Array
   int N;         // Anzahl items in PQ
   int size;      // Array-Groesse
   //int L, S;
public:
   PriorityQueue(int maxN); // Konstruktor
   bool empty() const;
// Test, ob Array leer
   bool isfull() const; // Test, ob Array voll
   void push_back(PQItem newItem); // Hinzufuegen eines
                                   // neuen Elementes
   int pop_max(); // Holen und Entfernen des
                  // Elementes mit maximaler Prioritaet
};
```

```
class PQItem{
public:
   int data;
   int prio;
};
```

PQ API mit 1-fach Verketteter Liste

```
class PriorityQueue {
private:
   // Zeiger auf 1. Listen-Element
   PQNode *pqhead;
public:
   PriorityQueue();      // Konstruktor
   bool empty() const;   // Test Liste leer

   // Hinzufuegen eines neuen Elementes
   void push_front(int item, int prio);

   // Element mit max. Prioritaet holen und entfernen
   int pop_max();
};
```

```
class PQNode{
public:
   int data;
   int prio;
   PQNode *next;
};
```

UNIVERSITY OF APPLIED SCIENCES

PQ Applikationen

1. Simulationssysteme (Zeitpunkte von Ereignissen)
2. Ressourcenverwaltung in Betriebssystemen (Prozessorzuteilung)
3. Numerische Berechnungen (Größter Fehler wird zuerst bearbeitet)
4. Basis für einen Sortieralgorithmus (alle Elemente erst einfügen, dann jeweils das größte Element entfernen)
5. Komprimierungsalgorithmen für Dateien
6. Suchalgorithmen für Graphen

Verkettete Liste (VL)

Definition (Verkettete Liste (Sedgewick, Wayne))

Eine verkettete Liste ist eine rekursive Datenstruktur, die wie folgt definiert ist: Sie ist entweder leer (null) oder besteht aus einer Referenz auf einen Knoten (head-Knoten), der ein generisches Element und eine Referenz auf eine verkettete Liste hält.

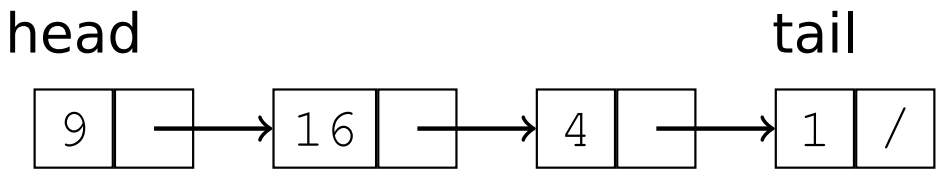
Definition (Verkettete Liste (Cormen et.al.))

Eine verkettete Liste ist eine Datenstruktur, bei der die Objekte in einer linearen Ordnung angelegt sind. Die Ordnung in einer verketteten Liste ist durch Zeiger auf die Vor- oder Nachfolger-Objekte gegeben.

Definition (Verkettete Liste)

Eine Menge von Datenobjekten, bei der jedes Objekt die Informationen enthält, um zum nächsten Element zu gelangen.

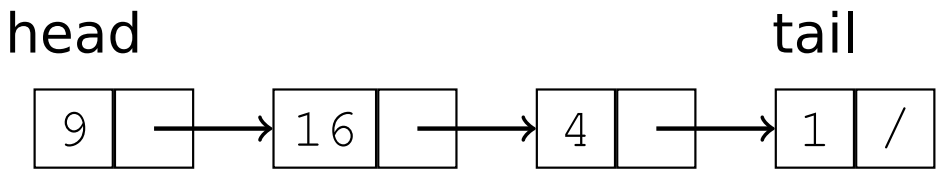
Einfach Verkettete Liste




```
class Node {
public:
  int  key;
  Node *next;
  Node(){
    key=0; next=0;}
};
```

Bei einer einfach verketteten Liste referenziert jeder Knoten nur auf seinen Nachfolgeknoten (next).

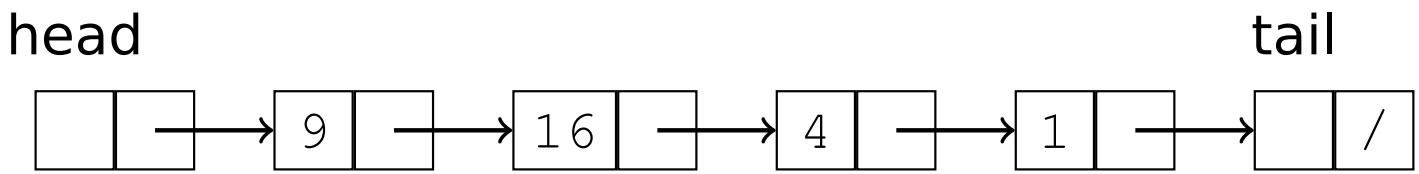
API Einfache VL




```
class Node {
public:
   int  key;
   Node *next;
   Node(){
      key=0; next=0;}
};
```

```
class List {
private:
    Node *head;
    Node *tail;
public:
    List(){head = 0; tail = 0;}
    ~List();
    void insertList(int key);
    bool deleteList(int key);
    int  searchList(int key);
};
```

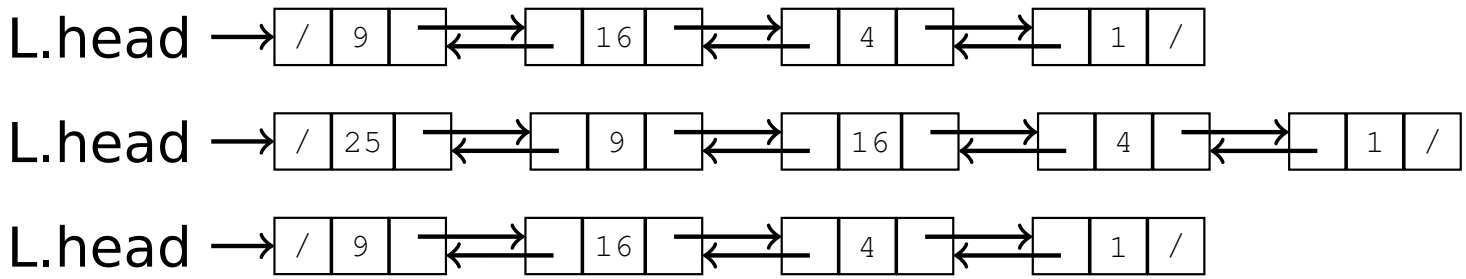
API Einfache VL mit expliziten Head- und Tail-Knoten




```
class Node {
public:
  int  key;
  Node *next;
  Node(){
    key=0; next=0;}
};
```

```
class List {
private:
  Node *head;
  Node *tail;
public:
  List(){
    head = new Node();
    tail = new Node();
    head->next = tail; }
  ~List();
  void insertList(int key);
  bool deleteList(int key);
  int  searchList(int key);
};
```

Doppelt Verkettete Liste




```
class Node {
public:
  int  key;
  Node *next;
  Node *prev;
  Node(){
    key=0;
    next=nullptr;
    prev=nullptr;}
};
```

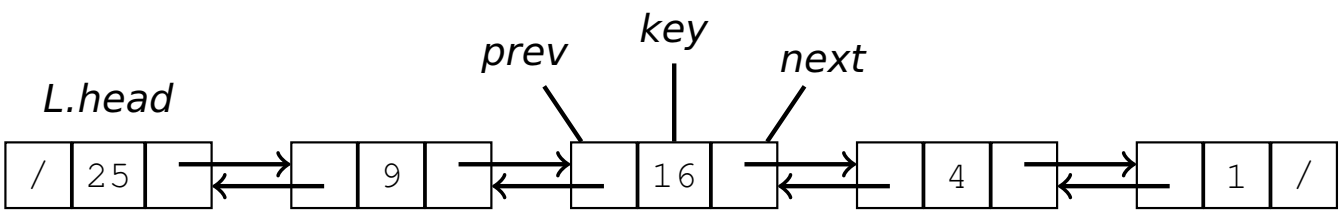
Bei einer doppelt verketteten Liste referenziert jeder Knoten auf seinen Vorgänger- (prev) und seinen Nachfolgeknoten (next).

Pseudocode List Search

Algorithm 5: Suche nach Element k in doppelt Verketteter Liste

Function *LIST_SEARCH(L, k)*
    $x \leftarrow L.head$
    while $x \neq NULL$ *and* $x.key \neq k$ **do**
        $x \leftarrow x.next$
    end
    return x
end

Annahme:
L sei doppelt verkettete Liste
mit *L.head* als ersten Listenknoten.

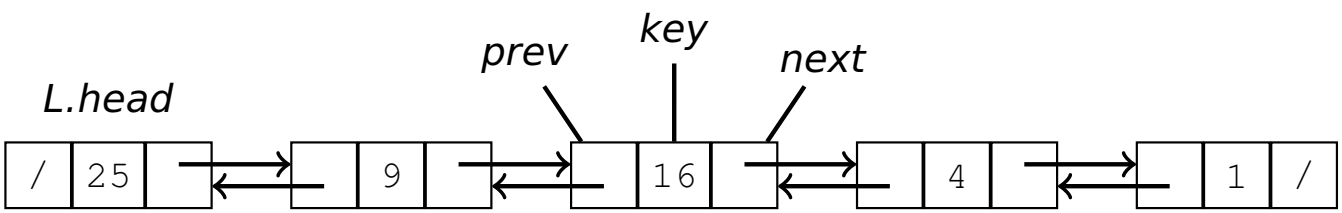


Pseudocode List Insert

Algorithm 6: Einfügen des Element x am Anfang der doppelt Verketteten Liste

Function *LIST_INSERT(L, x)*

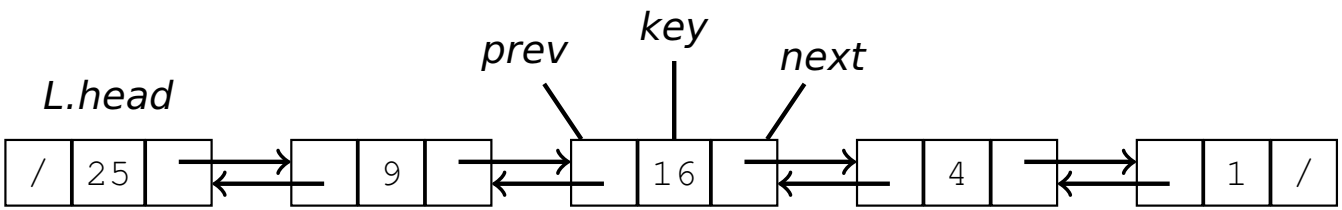
```
    x.next ← L.head
    x.prev ← NULL
    if L.head ≠ NULL then
        [L.head].prev ← x
    L.head ← x
end
```



Pseudocode List Delete

Algorithm 7: Löschen des Elements x aus der doppelt Verketteten Liste

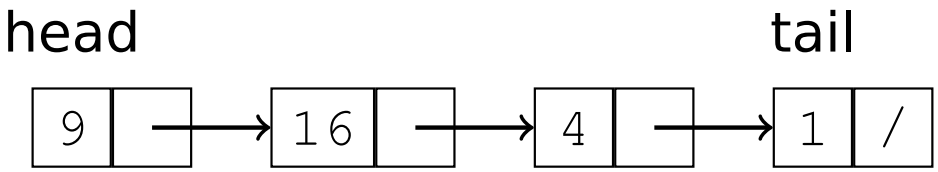
```
Function LIST_DELETE(L, x)
    if x.prev ≠ NULL then
        [x.prev].next ← x.next
    else
        L.head ← x.next
    end
    if x.next ≠ NULL then
        [x.next].prev ← x.prev
    delete x
end
```



Eigenschaften Verkettete Liste (VL)

- Eine verkettete Liste besteht aus miteinander verbundenen Knoten.
- Die Knoten einer verketteten Liste werden als eigene Datenobjekte repräsentiert mit einer Referenz auf den Nachfolger bei der 1-fach VL oder mit Referenzen auf den Vorgänger- und Nachfolgeknoten bei der 2-fach VL.
- Die Speicherbereiche für die Knoten einer VL sind im Speicher beliebig angeordnet.
- Es existieren ausgewiesene Anfangs- und Endknoten (head und ggfls. auch tail).
- Effiziente Neuanordnung der Knoten im Speicher durch Umbiegen von Referenzen möglich.
- Vorteil: Dynamisches Wachsen oder Schrumpfen, Größe der Liste muß nicht vorab bekannt sein.
- Nachteil: Kein direkter, sondern nur sequentieller Zugriff auf Listenelemente.

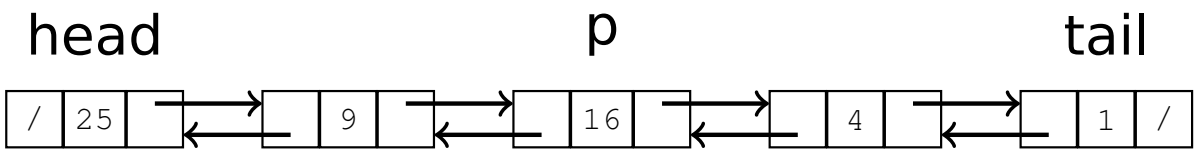
Queue mit einfach verketteter Liste



Entnahme am Listenanfang mit $O(1)$

Einfügen am Listenende mit $O(1)$

Übungen zur doppelt verketteten Liste



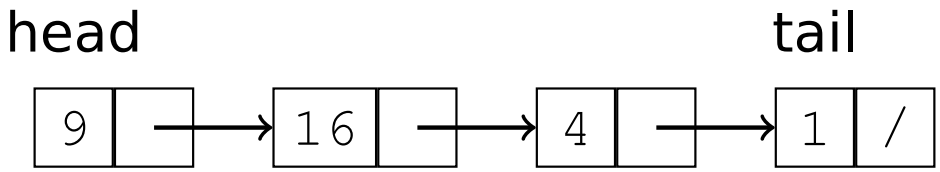
Übung 1

Schreiben Sie eine C++-Methode, die zu einer gegebenen Knotenreferenz *p* diesen Knoten *p* aus einer doppelt verketteten Liste löscht.

Übung 2

Schreiben Sie eine C++-Methode, die eine doppelt verkettete Liste durch Umbiegen der Referenzen aufwärts sortiert.

Stack mit Verketteter Liste



Entnahme am Listenanfang mit $O(1)$

Einfügen am Listenanfang mit $O(1)$

Inhalt - Kapitel 2: Elementare Datenstrukturen

- Kapitel 2 - Elementare Datenstrukturen
  - Felder (Array)
  - Stapel und Warteschlangen
    - Stack mit Array-Implementierung
    - Interface, Implementierung, Client
    - Warteschlange als Array mit Ringpuffer
  - Prioritätenwarteschlange
  - Verkettete Listen
    - Einfach Verkettete Liste
    - Doppelt Verkettete Liste
    - Stapel und Warteschlangen mit Verketteten Listen

Vielen Dank!

Prof. Ingrid Scholl
FH Aachen
Fachbereich für Elektrotechnik und Informationstechnik
Graphische Datenverarbeitung und Grundlagen der Informatik
MASKOR Institut
Eupener Straße 70
52066 Aachen
T +49 (0)241 6009-52177
F +49 (0)241 6009-52190
scholl@fh-aachen.de

www.fh-aachen.de