Einfache Datenstrukturen

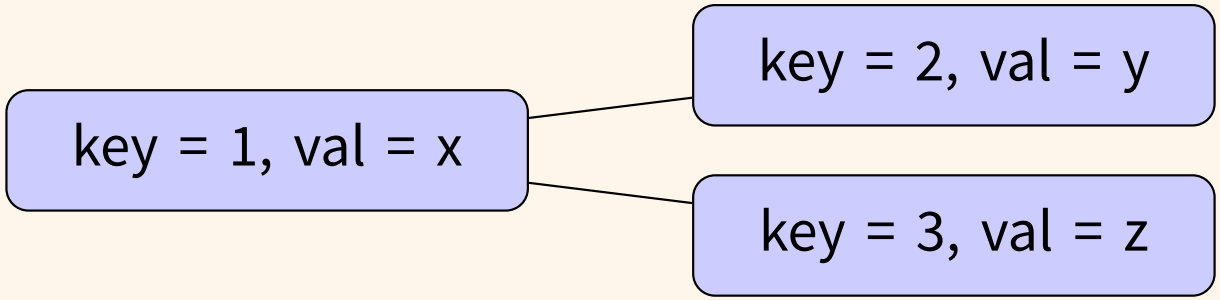
by

Dr. Günter Kolousek

Übersicht über Datenstrukturen

- Einfache Datenstrukturen
  - Liste
  - Stack
  - Queue, Deque
- Weitere Datenstrukturen
  - Ringbuffer
  - Priority Queue
  - Heap
  - Set, Bag
- Bäume
- Hashing
- Graphen

Repräsentation eines Baumes



- Liste

```
[1, "x",
  [2, "y", [], []],
  [3, "z", [], []]
]
```

Repräsentation eines Baumes – 2

- Dictionary

```
{
  ”key”: 1, ”val”: ”x”,
  ”left”: {
    ”key”: 2, ”val”: ”y”,
    ”left”: None, ”right”, None},
  ”right”: {
    ”key”: 3, ”val”: ”z”,
    ”left”: None, ”right”, None}
}
```

Repräsentation eines Baumes – 3

- Klasse

```
class Node:
    def __init__(self, key, val,
                 left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right


Node(1, ”x”, Node(2, ”y”), Node(3, ”z”))
```

Liste

- einfach verkettete Liste
  - singly linked list (sll)
  - jeder Knoten hat einen Zeiger `next`
- doppelt verkettete Liste
  - doubly linked list (dll)
  - jeder Knoten hat Zeiger `next` und `prev`
- Anzahl der Anker
  - 1 Anker: meist bei sll
  - 2 Anker: meist bei dll

(Übliche) Operationen

- `append` ... am Ende anhängen (C++ $\rightarrow$ `push_back`)
- `insert` ... an beliebiger Position einfügen
- `remove` ... an beliebiger Position löschen (C++ $\rightarrow$ `erase`)
- `get`, `set` ... verändern eines Elementes
  - wenn mit wahlfreiem Zugriff
- `empty`, `isEmpty` ... abfragen, ob leer
- `size` ... Größe abfragen

sll – Klasse

```
class Node:
    maxid = 0
    def __init__(self, data):
        Node.maxid += 1
        self.id = Node.maxid
        self.data = data
        self.next = None

    def __str__(self):
        return "Node({}, {})".format(
            self.data, self.next.id if self.next
                                    else None)

# use it:
head = None  # just an empty list
head = Node("maxi")  # with one single element
```

sll – Traversieren

```
def traverse(head, doit):
    curr = head
    while curr:
        doit(curr)
        curr = curr.next

# use it:
def print_data(node):
    print(node.data)

traverse(head, print_data)
```

sll – Suchen

```
# to remember: useful if you search for key...
def search(head, data):
    curr = head
    while curr:
        if curr.data == data:
            break
        curr = curr.next
    return curr

# use it
node = search(head, "maxi")
if node:
    print("found")
else:
    print("not found")
```

sll – Anhängen

```
def append(head, data):
    if head:
        curr = head
        while curr.next:
            curr = curr.next
        curr.next = Node(data)
    else:
        # list empty therefore create new head
        head = Node(data)
    return head
```

sll – Einfügen

```
# without checking preconditions:
#   - idx == 0 or list empty => insert at the beginning
#   - idx > length of list => append at the end
# must be true: idx >= 0
def insert(head, data, idx):
    curr = head
    if curr and idx != 0:
        i = 1
        while curr.next and i < idx:
            curr = curr.next
            i += 1
        # curr points at prev pos!    # without ".next.next":
        tmp = curr.next               # tmp = curr
        curr.next = Node(data)        # curr = Node(data)
        curr.next.next = tmp          # curr.next = tmp.next
                                      # tmp.next = curr
    else:
        # insert at the beginning (either empty or idx == 0)
        head = Node(data)
        head.next = curr
    return head
```

sll – Einfügen – 2

```
head = insert(head, "maxi", 0)
```

$\rightarrow$ per value (object reference per value)!

sll – Löschen

```
def remove(head, data):
    curr = head
    prev_curr = None
    while curr and curr.data != data:
        prev_curr = curr
        curr = curr.next
    if curr:
        # found, removing it...
        # ...relying on garbage collection!
        if prev_curr:
            prev_curr.next = curr.next
        else:
            head = curr.next
    return head
```

Stack

- Stack (Stapel, Kellerspeicher) ist eine Sonderform der Liste
- Prinzip: LIFO
- Übliche Operationen:
  - push, append
  - pop
  - top, peek
  - empty, isEmpty
- Anwendungen: Methodenaufruf, Back-Button,…

Stack – Anwendungen

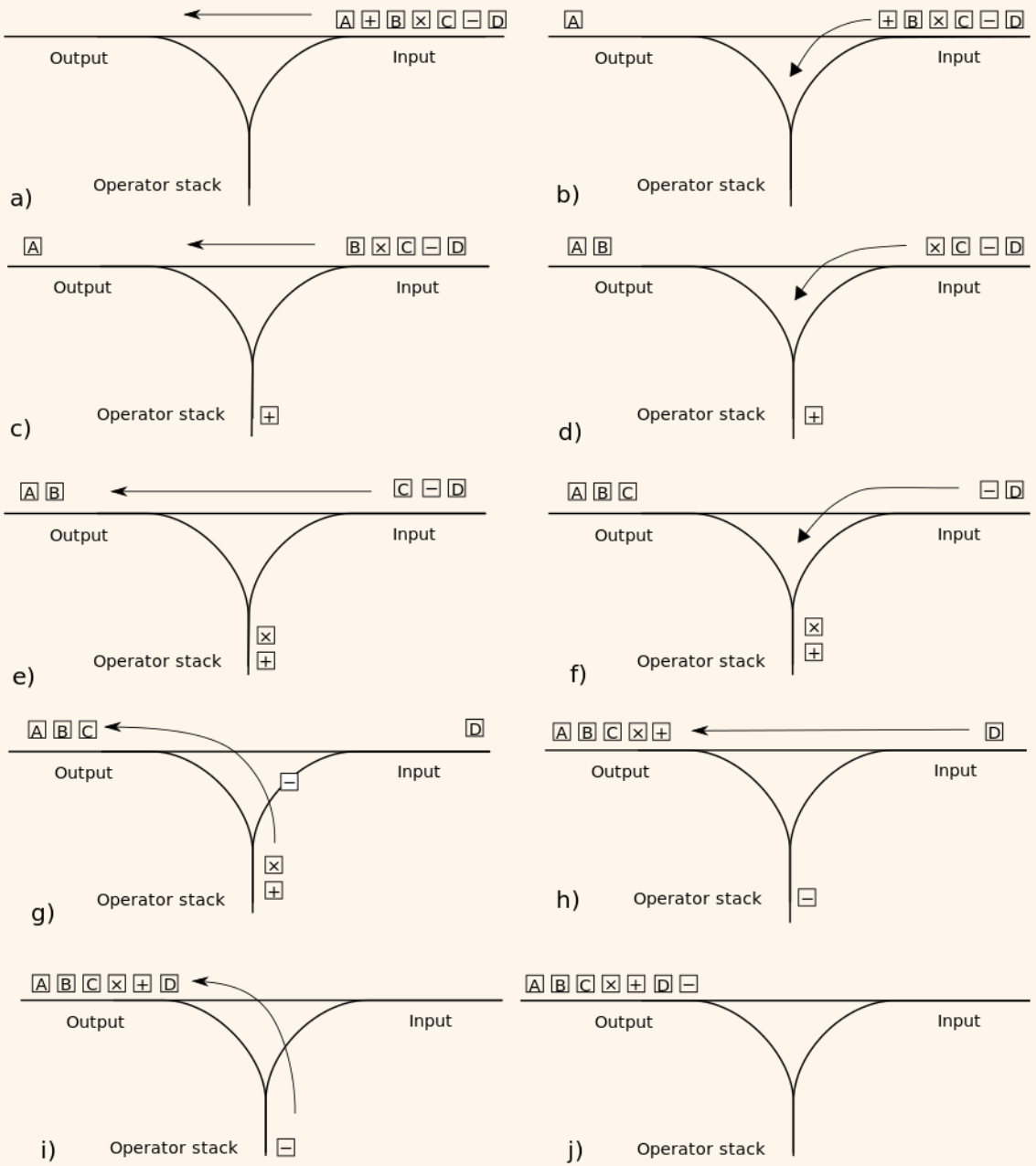
1. Abarbeitung von Postfix - Ausdrücken
   - Ausdruck von links nach rechts lesen
   - Gelesenes Symbol ist Operand, dann auf Stack
   - Gelesenes Symbol ist n-stelliger Operator, dann n Operanden vom Stack, auswerten und Ergebnis auf Stack
2. Infix nach Postfix
   - Linke Klammern: ignorieren
   - Rechte Klammern: pop und print
   - Operator: push
   - Operand: print
   - Am Ende:
     - Stack abräumen und alles: print

Shunting-yard Algorithmus

von Dijkstra, ”Rangierbahnhof”, Infix → Postfix

```
for tok in tokens:
    if isoperand(tok):
        output += tok  # print
    elif isoperator(tok):
        while stack and stack[-1] != "(" and \
                priority(stack[-1]) >= priority(tok):
            output += stack.pop()
        stack.append(tok)  # push
    elif tok == "(":
        stack.append(tok)
    elif tok == ")":
        while stack[-1] != "(":
            output += stack.pop()  # pop and print
        stack.pop()  # ignore left parenthesis
while stack:
    output += stack.pop()  # pop and print
```

Shunting-yard Algorithmus – 2



Quelle: Wikipedia

Shunting-yard Algorithmus – 3

- Arbeitet mit
  - (und)
  - binären linksassoziativen Infixoperatoren
  - Prioritäten
- Verbesserungen
  - Unterscheidung unäre und binäre Operatoren
    - z.B. -1 vs. $5-3$
    - kein Problem, wenn Blanks als Trennzeichen
  - Postfixoperatoren
    - z.B. $3\,!$
  - rechtsassoziative Operatoren
  - Funktionen

Queue

- Queue ist eine Sonderform der Liste
- Prinzip: FIFO
- Übliche Operationen:
  - put, push, enqueue
  - take, pop, dequeue
  - front, back
  - empty, isEmpty
- Anwendungen: Warteschlange (z.B. Drucker), Prozessbearbeitung

Deque

- Double-ended queue, **d**ouble-**e**nded **que**ue
- Prinzip: wie Queue, aber beide Seiten
- Übliche Operationen:
  - append, `push_back` addLast
  - appendleft, addFirst
  - pop, removeLast
  - popleft, removeFirst
  - front, getFirst
  - back, getLast
  - empty, isEmpty

Ringpuffer

- engl. ringbuffer, circular buffer
- Puffer hat fixe Größe! Verwendung als Ring!
  - Anwendungen: Multimedia, Flugschreiber,…
  - Optimal für Queue mit **fixer** Größe
- Schreiben am Ende, Lesen am Anfang
  - Wenn voll, dann
    - alte Daten werden überschrieben oder
    - Fehler oder Exception
- Es werden prinzipiell 3 Angaben benötigt:
  - Adresse und Größe des Arrays `arr`
  - Leseposition (Index, Adresse)
  - Schreibeposition (Index, Adresse)

Ringpuffer – Implementierung 1

- Lese- (`read_idx`) und Schreibindex (`write_idx`)
  - leer vs. voll
    - `read_idx == write_idx`, dann leer
    - `read_idx == (write_idx + 1) % SIZE`, dann voll
  - Schreiben: Überprüfen, ob nicht voll, dann Schreiben und Index imkrementieren (modulo SIZE)
  - Lesen: Überprüfen, ob leer, dann Lesen und Index imkrementieren (modulo SIZE)
  - Nachteil: Speicher wird nicht vollständig genutzt, Zugriff über Indizes

Ringpuffer – Implementierung 2

- Schreibindex `write_idx` und Anzahl der Elemente `fill_cnt`
  - beim Schreiben: Überprüfen, ob nicht voll, dann schreiben und Index inkrementieren (modulo SIZE) und `fill_cnt` inkrementieren
  - beim Lesen
    - Überprüfen, ob leer
    - dann:

```
read_idx = write_idx - fill_cnt
if read_idx < 0:
    read_idx += SIZE
fill_cnt -= 1
```

  - Nachteil: Rechenoperationen beim Lesen! Zugriff über Indizes

Ringpuffer – Implementierung 3

- Schreibepointer `write_pos`, Lesepointer `read_pos` und Anzahl der Elemente `fill_cnt` (oder auch ein Flag `empty`)
  - beim Schreiben
    1. Überprüfen, ob voll, dann schreiben und `write_pos++`
    2. Wenn `write_pos == arr + SIZE`, dann `write_pos = arr`
    3. `++fill_cnt`
  - beim Lesen
    1. Überprüfen, ob leer, dann lesen und `read_pos++`
    2. Wenn `read_pos == arr + SIZE`, dann `read_pos = arr`
    3. `--fill_cnt`
  - Vorteil: Zeigerarithmetik (kein Zugriff über Indizes), Vollständige Nutzung (im Vergleich zu Implementierung 1)
  - Nachteil: zusätzliche Variable

Priority Queue

- Wie Queue, aber
  - jeder Eintrag hat Priorität
  - nächstes Element ist Element mit höchster Priorität
- Implementierungen
  - als unsortierte Liste $\rightsquigarrow$ immer durchsuchen!
  - als sortierte Liste
  - als Heap

Heap

- Heap (Halde, Haufen): auf Baum basierende Datenstruktur
- Max-Heap: Elternknoten immer $\geq$ als Kindknoten
- Min-Heap: Elternknoten immer $\leq$ als Kindknoten
- binärer Heap: Heap basierend auf binären Baum
- binärer Max-Heap
  - Eine Folge von $F = k_1, k_2, ..., k_n$ von Schlüsseln, wenn $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$, soferne $2i \leq n$ bzw. $2i + 1 \leq n$ ist, wird als (binärer) Max-Heap bezeichnet
  - Ein (binärer) Max-Heap wird oft als Array implementiert
- Übliche Operationen: `insert`, `extract`
- Anwendungen: $\rightsquigarrow$ Heapsort, priority queues (z.B. Jobqueue)

Heap – 2

Max-Heap

- `insert(arr, key)`
  1. key am Ende anhängen (= neuer Knoten)
  2. Wenn Elternknoten kleiner als neuer Knoten:
     - dann: mit aktuellen Knoten vertauschen und mit Schritt 2 weitermachen
     - anderenfalls: fertig
- `extract(arr)`
  1. Wurzel durch letzten Knoten ersetzen
  2. Wenn ersetzter Knoten kleiner als Kindknoten:
     - dann mit größeren der beiden Kindknoten vertauschen und mit Schritt 2 weitermachen
     - anderenfalls: fertig

Bag

- $\rightsquigarrow$ Multimenge
- Python: `collections.Counter` kann als Bag verwendet werden

```
>>> x = Counter({”a”, ”b”})
>>> x
Counter({'a': 1, 'b': 1})
>>> x[”b”] += 1
>>> y = Counter({”a”, ”b”})
>>> y[”a”] += 2
>>> x & y
Counter({'a': 1, 'b': 1})
>>> x | y
Counter({'a': 3, 'b': 2})
>>> x - y
Counter({'b': 1})
```