Mobile Computing und Softwareengineering
SEN

Felix Hofinger

April 2022

Contents

1 Listen

Vorteile gegenüber Arrays:

- wachsen und schrumpfen
- einfaches einfügen und löschen

Nachteile gegenüber Arrays:

- kein indizierter Zugriff

1.1 Lineare List

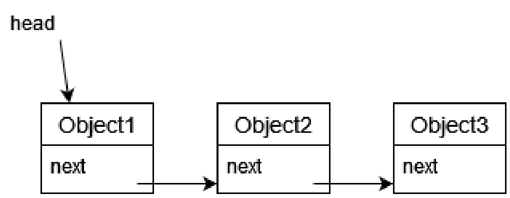


Figure 1: Aufbau einer Liste

```
struct Node {
    struct Node* next;
    int value;
}

struct List {
    struct Node* head;
}

int main() {
    struct List* list = (struct List*) malloc(sizeof(struct List));
    list->head = NULL;
}
```

1.1.1 Einfügen am Listen Anfang

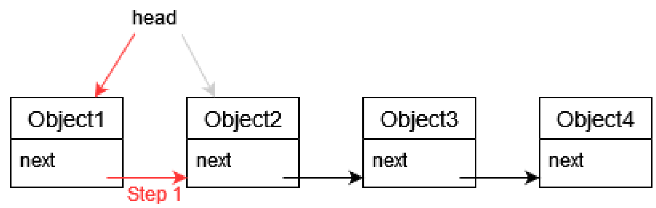


Figure 2: Einfügen am Listen Anfang

```
void prepend(struct List* list, int value) {
    struct Node* n = (struct Node*) malloc(sizeof(struct Node));
    n->value = value;
    n->next = list->head;
    list->head = n;
}
```

1.1.2 Einfügen am Listen Ende

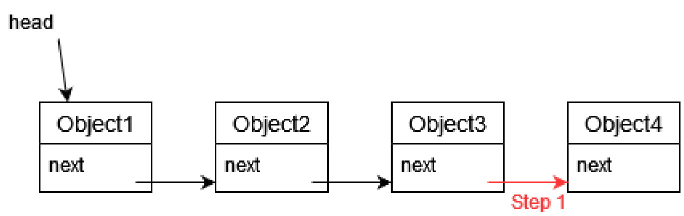


Figure 3: Einfügen am Ende der Liste

```
void append(struct List* list, int value) {
    struct Node* n = (struct Node*) malloc(sizeof(struct Node));
    n->value = value;
    n->next = NULL;

    if (list->head == NULL) {
        list->head = n;
        return;
    }

    struct Node* p = head;
    while (p->next != NULL) {
        p = p->next;
```

```
    }
    p->next = n;
}
```

1.1.3 Suchen

```
struct Node* search(struct List* list, int value) {
    struct Node* p = list->head;
    while (p != NULL && p->value != value) {
        p = p->next;
    }
    return p;
}
```

1.1.4 Löschen am Listen Anfang

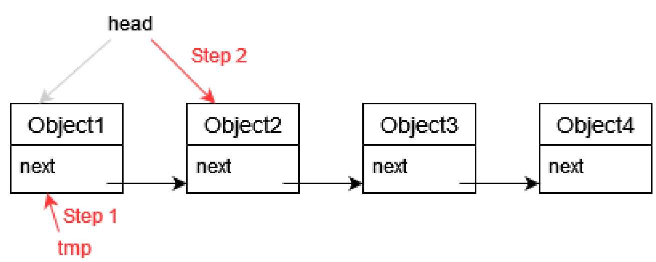


Figure 4: Löschen am Anfang der Liste

```
struct Node* remove(struct List* list) {
    if (list->head != NULL) {
        struct Node* tmp = list->head;
        list->head = list->head->next;
        free(tmp);
    }
}
```

1.1.5 Löschen eines Knotens mit Schlüssel

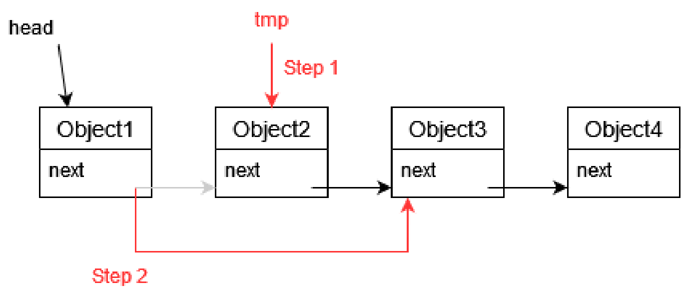


Figure 5: Löschen in der Mitte der Liste

```
void remove(struct List* list, int value) {
    struct Node* p = list->head;
    struct Node* prev = NULL;

    while (p != NULL && p->value != value) {
        prev = p;
        p = p->next;
    }

    if (p == NULL) {
        return; // Liste leer oder Wert nicht gefunden
    }

    if (p == list->head) {
        list->head = list->head->next;
        free(p);
    } else {
        prev->next = p->next;
        free(p);
    }
}
```

1.1.6 Eine Liste Reversen

```
void reverse() {
    if (count() <= 0) return;
    node* n = head, * new_head = NULL;
    int i;
    for (i = 0; n != NULL; i++) {
        if (new_head == NULL) {
```

```
            new_head = n;
        } else {
            node* nn = n->next;
            n->next = new_head;
            new_head = n;
            n = nn;
        }
    }

    n = new_head;
    for (int j = 0; j < i; j++) {
        n = n->next;
    }
    n->next = NULL;

    head = new_head;
}
```

1.1.7 Sortieren einer Liste

```
void sort() {
    int len = count();

    for (int j = 0; j < len; j++) {
        for (int i = 0; i < len - j - 1; i++) {
            node* n = head;
            for (int k = 0; k < i; k++) {
                n = n->next;
            }

            node* npp = n->next;
            if (n->data > npp->data) {
                int temp = n->data;
                n->data = npp->data;
                npp->data = temp;
            }
        }
    }
}
```

1.2 Sortierte Liste

1.2.1 Einfügen (ohne doppelte)

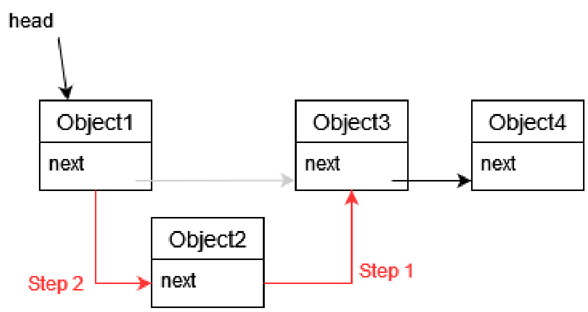


Figure 6: Einfügen in der Mitte einer Liste

```
void insert(struct List* list, int value) {
    struct Node* p = list->head;
    struct Node* prev = NULL;

    while (p != NULL && p->value < value) {
        prev = p;
        p = p->next;
    }

    if (p == NULL || p->value != value) {
        struct Node* n = (struct Node*) malloc(sizeof(struct Node));
        n->value = value;
        n->next = p;
        if (p == list->head) {
            list->head = n;
        } else {
            prev->next = n;
        }
    }
}
```

2 Stack

Prinzip: LIFO (Last in, First out)

2.1 Stack mit Array

```
struct Stack {
    int* data;
    int size;
    int top;
}

struct Stack* init(int stacksize) {
    struct Stack* s = (struct Stack*) malloc(sizeof(struct Stack));
    s->data = (int *) malloc(stacksize * sizeof(int));
    s->size = stacksize;
    s->top = 0;
    return s;
}

void push(struct Stack* s, int value) {
    if (s->top < s->size) {
        s->data[s->top] = value;
        s->top++;
    } else {
        exit(-1); // Stack voll!!
    }
}

int pop(struct Stack* s) {
    if (s->top > 0) {
        s->top--;
        return s->data[s->top];
    } else {
        exit(-1);
    }
}
```

2.2 Stack mit Linked List

```
struct Node {
    struct Node* next;
    int data;
}
```

```c
struct Stack {
    struct Node* head;
}

struct Stack* init(int stacksize) {
    struct Stack* s = (struct Stack*) malloc(sizeof(struct Stack));
    s->head = NULL;
    return s;
}

void push(struct Stack* s, int v) {
    struct Node* n = (struct Node*) malloc(sizeof(struct Node));
    n->data = value;
    n->next = s->head;
    s->head = n;
}

int pop(struct Stack* s) {
    if (s->head == NULL) {
        exit(-1);
    }

    int value = s->head->data;
    struct Node* n = s->head;
    s->head = s->head->next;
    free(n);
    return value;
}
```

3 Queue

Prinzip: FIFO (First in, First out)

3.1 Queue als Array

```c
typedef struct {
    int* queue;
    int head;
    int tail;
    int elements;
    int len;
} Queue;
```

```
Queue* createQueue(int startsize) {
    Queue *q = (Queue *) malloc(sizeof(Queue));
    q->data = (int *) malloc(sizeof(int) * startsize);
    q->capacity = startsize;
    q->size = 0;
    q->tail = 0;
    q->head = 0;
    return q;
}

void put(Queue *q, int v) {
    if (q->size >= q->capacity) {
        printf("Not enough capacity!\n");
        return;
    }
    q->data[q->tail] = v;
    q->size++;
    q->tail = (q->tail + 1) % q->capacity;
}

int get(Queue *q) {
    if (q->size <= 0) {
        printf("Queue empty!\n");
        return NULL;
    }
    int v = q->data[q->head];
    q->head = (q->head + 1) % q->capacity;
    return v;
}

int peek(Queue *q) {
    if(q->size > 0){
        return q->data[q->head];
    }
    printf("Queue empty\n");
    return -1;
}
```

3.2 Queue als List

```
typedef struct {
    Node* next;
    int value;
} Node;
```

```
typedef struct {
    Node* head;
    Node* tail;
} Queue;

Queue* createQueue() {
    Queue* q = (Queue*) malloc(sizeof(Queue));
    q->head = NULL;
    q->tail -> NULL;
    return q;
}

void put(Queue* q, int value) {
    Node* n = (Node*) malloc(sizeof(Node));
    n->next = NULL;
    n->value = value;

    if (q->tail == NULL) {
        q->head = n;
        q->tail = n;
    } else {
        q->tail->next = n;
    }
}

int get(Queue* q) {
    Node* tmp = q->head;
    int v = tmp->value;
    q->head = q->head->next;
    free(tmp);
    return v;
}
```

4 Komplexität von Algorithmen

TODO: add beispiel from P5b, P6f, P6b

4.1 Beispiel

4.1.1 Verbesserung #1

4.1.2 Verbesserung #2

4.2 Analyse

4.3 Typische Komplexitätsfunktionen von Algorithmen

1	konstant	Jede Anweisung wird einmal ausgeführt. Idealzustand.
log(n)	logarithmisch	Basis 2 -> vierfache Datenmenge, doppelte Ressourcen
n	linear	direkt proportional
n * log(n)	n log n	zwischen n und n^2
n^2	quadratisch	wächst quadratisch nur für kleine Probleme anwendbar
n^3	kubisch	wächst kubisch nur für sehr kleine Probleme anwendbar
2^n	exponentiell	praktisch kaum verwendbar

Table 1: Komplexität von Funktionen

4.4 O-Notation

TODO: add o-notation text

4.4.1 Beispiel #1

4.4.2 Beispiel #2

5 Bäume

Jeder Knoten hat nicht nur einen, sondern mehrere direkte Nachfolger.

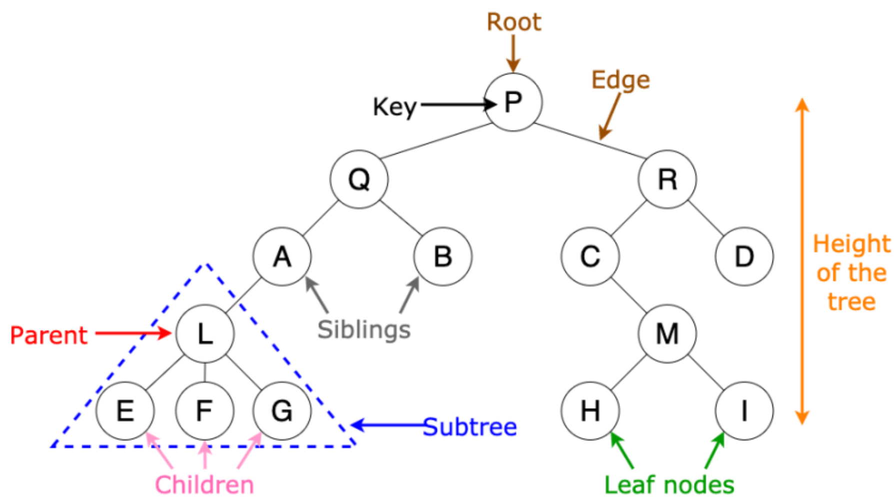


Figure 7: Aufbau eines Baumes

5.1 Binärbäume

Jeder Knoten hat maximal 2 Söhne (Children).

Verboten:

- Zyklen
- zwei Väter (Parents)

5.2 Binäre Suchbäume

```
class Node {
    int value;
    Node left;
    Node right;
    Node(int value) {
        this.value = value;
        left = null;
        right = null;
    }
}
```

```
class Tree {
    Node root;

    Tree() {
        root = null;
    }

    Node search(int value) {
        ...
    }

    void insert(int value) {
        ...
    }

    Node delete(int value) {
        ...
    }
}
```

5.2.1 Suchen

Problemgröße: Anzahl der Knoten bzw. Anzahl der zu durchsuchenden Werte.
O(log n): Bei 15 Knoten gibt es maximal 4 Such-Schritte.

5.2.1.1 Iterativ

```
Node search(int value) {
    Node p = root;
    while (p != null) {
        if (p.value == value) return value;
        if (value < p.value) p = p.left;
        else p = p.right;
    }
    return null; // Value not found
}
```

5.2.1.2 Rekursiv

```
Node search(int value) {
    return search(root, value);
}

static Node search(Node p, int value) {
    if (p == null || p.value == value) return p;
```

```
    else if (value < p.value) return search(p.left, value);
    else return search(p.right, value);
}
```

5.2.2 Einfügen

Wichtig: Immer ganz unten als Blatt (Leaf Node) einfügen.

5.2.2.1 Iterativ

```
void insert(int value) {
    Node p = root;
    Node father = null;
    while (p != null) {
        father = p;
        if (value < p.value) p = p.left;
        else p = p.right;
    }

    Node n = new Node(value);
    if (father == null) {
        root = n;
    } else if (value < father.value) {
        father.left = n;
    } else {
        father.right = n;
    }
}
```

5.2.2.2 Rekursiv

```
void insert(int value) {
    root = insert(root, value);
}

static Node insert(Node p, int value) {
    if (p == null) p = new Node(value);
    else if (value < p.value) p.left = insert(p.left, value);
    else p.right = insert(p.left, value);
    return p;
}
```

5.2.3 Löschen

TODO: add image of the 4 different cases. P9b

```
Node delete(int value) {
    Node father = null;
    Node p = root;
    while (p != null && p.value != value) {
        father = p;
        if (value < p.value) p = p.left;
        else p = p.right;
    }

    if (p != null) {
        Node x;
        if (p.right == null) {
            x = p.left;
        } else if (p.right.left == null) {
            x = p.right;
            x.left = p.left;
        } else {
            Node xf = p.right;
            x = xf.left;
            while (x.left != null) {
                xf = x;
                x = x.left;
            }

            xf.left = x.right;
            x.left = p.left;
            x.right = p.right;
        }

        if (p == root) root = x;
        else if (value < father.value) father.left = x;
        else father.right = x;

        p.left = null;
        p.right = null;
    }

    return p;
}
```

5.3 Traversieren von Bäumen

TODO: add example tree. P10f

Pre-Order	Post-Order	In-Order
Wurzel/links/rechts	links/rechts/Wurzel	links/Wurzel/rechts
+*24/93	24*93/+	2*4+9/3

Table 2: Richtungen um einen Baum zu durchlaufen.

TODO

5.4 Balancieren von Bäumen

5.4.1 Tree to Vine

5.4.1.1 Beispiel

5.4.2 Vine to Tree

TODO

6 Graphen

6.1 Speicherdarstellung

6.1.1 Adjazenzliste

6.1.2 Adjazenzmatrix

6.2 Depth-First-Search (DFS)

6.3 Breath-First-Search (BFS)

6.4 Minimal-Spanning-Tree (MST)

6.5 Shortest Path (Dijkstra)

7 Hashing

Motivation:

Schlüssel	**Wert**
"Tom"	0664 12345
"Hans"	0699 98765
"Paul"	...

Table 3: Hash-Tabellen Example

val = tab["Tom"] => val = tab[42]

7.1 Hashing-Funktionen

Abbildung von großem Wertebereich (z.B.: Menge aller Dinge) auf einen kleinen Bereich (z.B.: integer).

7.1.1 Ziel

- gleichmäßige Streuung (wenig Kollisionen)
- einfach und schnell zu berechnen

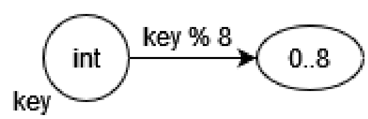


Figure 8: Einfache Hashing-Funktion

7.1.2 Langer Schlüssel (z.B.: String)

TODO exmaple table P15f

```
int hash(String key) {
    int adr = 0;
    for (int i = 0; i < key.length(); i++)
        adr = (2 * adr + key.charAt(i)) % tabSize;
    return adr;
}
```

-> sehr gute Hash-Funktion

-> ABER bei langen Schlüssel langsam

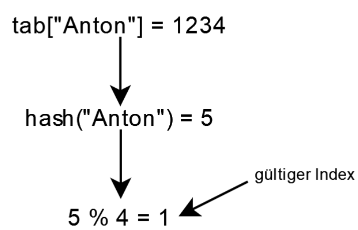
7.1.3 Alternative

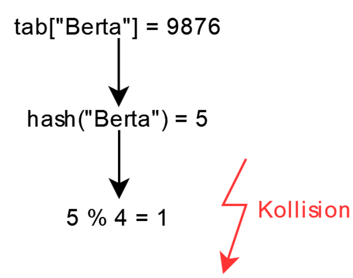
```
h = (key.charAt(0) * PRIMZAHL + key.charAt(key.length() - 1)
     + key.length()) % WERTEBEREICH;
```

```
hash("Anton") = (65 * 17 + 110 + 5) % 499 = 222
```

7.2 Kollisionsstrategie

Hashfunktion: h = str.length()   (<- sehr schlechte Hashfunktion)





7.2.1 Seperate Chaining

7.2.2 Linear Probing

7.2.3 Quadratic Probing

8 Datenbanken

8.1 Grundsätze

8.2 ER-Modell

- Entity: Dinge / Gegenstände / Objekte / ... (in Java: Klassen)

- Relationship: Beziehung zwischen Dingen (Entities)
- Redundanzfreie Datenspeicherung: Information kommt nur an einer einzigen Stelle in der Datenbank vor
- Datenkonsitenz: Daten müssen eindeutige Informationen darstellen

8.3 Wichtige Begriffe

- Entität (Entity): Tabellenname, Themenkreis
- Entitätsmenge: Datensätze
- Relation: Tabelle = Entität und Entitätsmenge
- Tuple: Datensatz
- Attribute: Spaltennamen
- Attribute-Value: Wert in einer Spalte
- Domain: Typ (Datentyp)

8.4 Beziehungen

1	einfache Assoziation	genau ein Tupel
c	konditionelle Assoziation	kein oder genau ein Tupel
m	multiple Assoziation	mindestens ein Tupel
mc	mutiple-konditionelle Assoziation	beliebig viele Tupel

Table 4: Caption

8.4.1 1-1 Beziehung

8.4.2 1-C Beziehung

8.4.3 1-M Beziehung

8.4.4 C-C Beziehung

8.4.5 C-M Beziehung