Programmieren 1

Auditorium Exercise 10

Human-Computer Interaction Group

Jan Feuchter, Michael Rohs
programmieren1@hci.uni-hannover.de

bitte teilnehmen

LEHREVALUATION

Weiterer Verlauf Übungen

	Ausgabe	Abgabe	Besprechung
Ü10	16.12.	22.12.	09.01.-12.01.
Ü11	22.12.	12.01.	13.01.-19.01.
Ü12	13.01.	19.01.	23.01.-26.01.
Ü13 (optional, Klausurvorbereitung)	20.01.	-	-
letzte Vorlesung 27.01.			

Klausur

- Zeitraum: 14.–16.3.
- Im Rechnerraum: Hauptgebäude F411
  - Hier finden auch die Tutorien statt
- Zeitslots (2 Stunden) über die 3 Tage verteilt
  - Zuteilung erfolgt per Stud.IP

ASSIGNMENT 9

Matrix

```c
struct Matrix {
int rows; // number of rows
int cols; // number of columns
double** data; // array of pointers, length: rows
               // a row is an array of doubles, length: cols
};
typedef struct Matrix Matrix;
```

Leibniz Universität Hannover

ASSIGNMENT 10

filesystem.c

```c
typedef enum {
    NT_DIR,
    NT_FILE,
} NodeType;

#define MAX_NAME_LEN 63

typedef struct Node Node;
typedef struct Entry Entry;

struct Entry {
    Node* node;
    Entry* next;
};
```

```c
struct Node {
    char name[MAX_NAME_LEN + 1];
    NodeType type;
    union {
        struct {
            Entry* entries; // list
        } dir;
        struct {
            void* contents; // binary data array
            int length; // number of bytes
        } file;
    };
};
```

filesystem.c – Alternative Data Definition (not used in assignment)

```c
typedef enum {
    NT_DIR,
    NT_FILE,
} NodeType;

#define MAX_NAME_LEN 63

typedef struct Node Node;
```

```c
struct Node {
    char name[MAX_NAME_LEN + 1];
    NodeType type;
    union {
        struct {
            Node* entries; // list
        } dir;
        struct {
            void* contents; // binary data array
            int length; // number of bytes
        } file;
    };
    Node* next; // next directory entry
};
```

filesystem.c – Preconditions/Asserts or Parameter Checks?

```
int file_read(Node* file, void* buffer, int length) {
    if (file == NULL || buffer == NULL || length <= 0) return 0;
    if (file->type != NT_FILE) return 0;
    int file_length = file->file.length;
    int n = file_length < length ? file_length : length;
    assert("contents exists", file->file.contents != NULL);
    memcpy(buffer, file->file.contents, n);
    return n;
}
```

filesystem.c – What about syntax errors in paths?

```
//a.txt
```

file a.txt in in root directoy?

interpreted as /a.txt

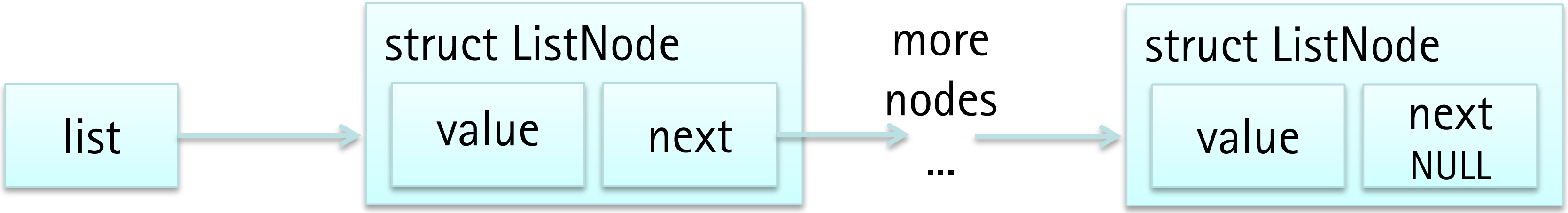
file a.txt is in directory with empty name below root directory?

interpreted as ε/ε/a.txt

Warum Listen? Warum nicht einfach Arrays?

Linked Lists

- Linked lists:



list

struct ListNode

value

next

more nodes …

struct ListNode

value

next NULL

Warum gibt append einen Node-Zeiger zurück?

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {









}
```

append_list – iterativ

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {
   if (list == NULL) { // empty list

   } else { // non-empty list





   }
}
```

append_list – iterativ

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {
   if (list == NULL) { // empty list
      return new_node(value, NULL);
   } else { // non-empty list





   }
}
```

append_list – iterativ

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {
   if (list == NULL) { // empty list
      return new_node(value, NULL);
   } else { // non-empty list




      return list;
   }
}
```

append_list – iterativ

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {
   if (list == NULL) { // empty list
      return new_node(value, NULL);
   } else { // non-empty list
      Node* n = list;
      while (n->next != NULL) n = n->next; // find last element


      return list;
   }
}
```

append_list – iterativ

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {
   if (list == NULL) { // empty list
      return new_node(value, NULL);
   } else { // non-empty list
      Node* n = list;
      while (n->next != NULL) n = n->next; // find last element
      assert("on last element", n != NULL && n->next == NULL);

      return list;
   }
}
```

append_list – iterativ

```
// Adds an element to the end of the list.
Node* append_list(Node* list, int value) {
   if (list == NULL) { // empty list
      return new_node(value, NULL);
   } else { // non-empty list
      Node* n = list;
      while (n->next != NULL) n = n->next; // find last element
      assert("on last element", n != NULL && n->next == NULL);
      n->next = new_node(value, NULL);
      return list;
   }
}
```

Aufwand für Listenoperationen

new_node: create a list node (heap allocation)	1 step
free_list: release dynamic memory	n steps
print_list: print contents	n steps
length_list: number of elements	n steps
prepend_list: add element to front of list	1 step
append_list: add element to end of list	n steps[1]
insert_list: insert an element at a certain position	i steps
remove_list: remove the element at a certain position	i steps
copy_list: copy each node to get two independent lists	n steps

[1] can be improved to 1 step

Warum gibt efficient_append nichts zurück?

```
void efficient_append_list(Lst* list, int value) {






}
```

```
struct Node {
    double value;
    struct Node *next;
};
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

Efficiently Adding at the End of a List

```
void efficient_append_list(Lst* list, int value) {
    require("list head exists", list != NULL);








}
```

```
struct Node {
    double value;
    struct Node *next;
};
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

Efficiently Adding at the End of a List

```
void efficient_append_list(Lst* list, int value) {
    require("list head exists", list != NULL);

    if (list->first == NULL) { // empty list, first and last change



    } else { // non-empty list, only last changes



    }
}
```

```
struct Node {
    double value;
    struct Node *next;
};
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

Efficiently Adding at the End of a List

```
void efficient_append_list(Lst* list, int value) {
    require("list head exists", list != NULL);
    Node* n = new_node(value, NULL);
    if (list->first == NULL) { // empty list, first and last change
        list->first = n;
        list->last = n;
    } else { // non-empty list, only last changes


    }
}
```

```
struct Node {
    double value;
    struct Node *next;
};
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

Efficiently Adding at the End of a List

```
void efficient_append_list(Lst* list, int value) {
    require("list head exists", list != NULL);
    Node* n = new_node(value, NULL);
    if (list->first == NULL) { // empty list, first and last change
        list->first = n;
        list->last = n;
    } else { // non-empty list, only last changes
        list->last->next = n;
        list->last = n;
    }
}
```

```
struct Node {
    double value;
    struct Node *next;
};
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

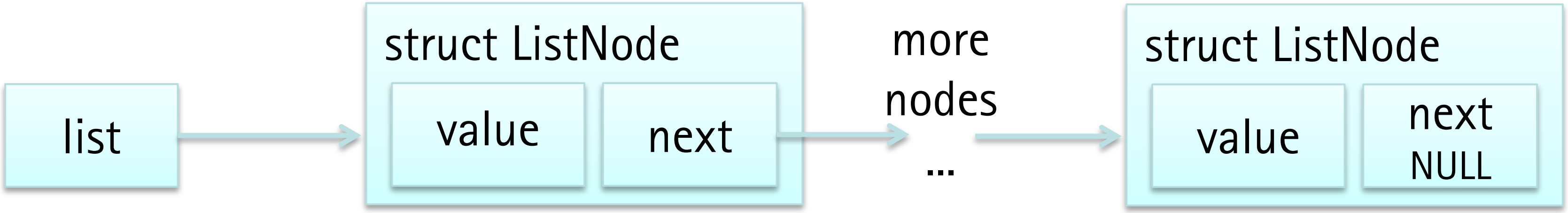
Unterschied zwischen Listen und Binärbäumen?

- Linked lists:

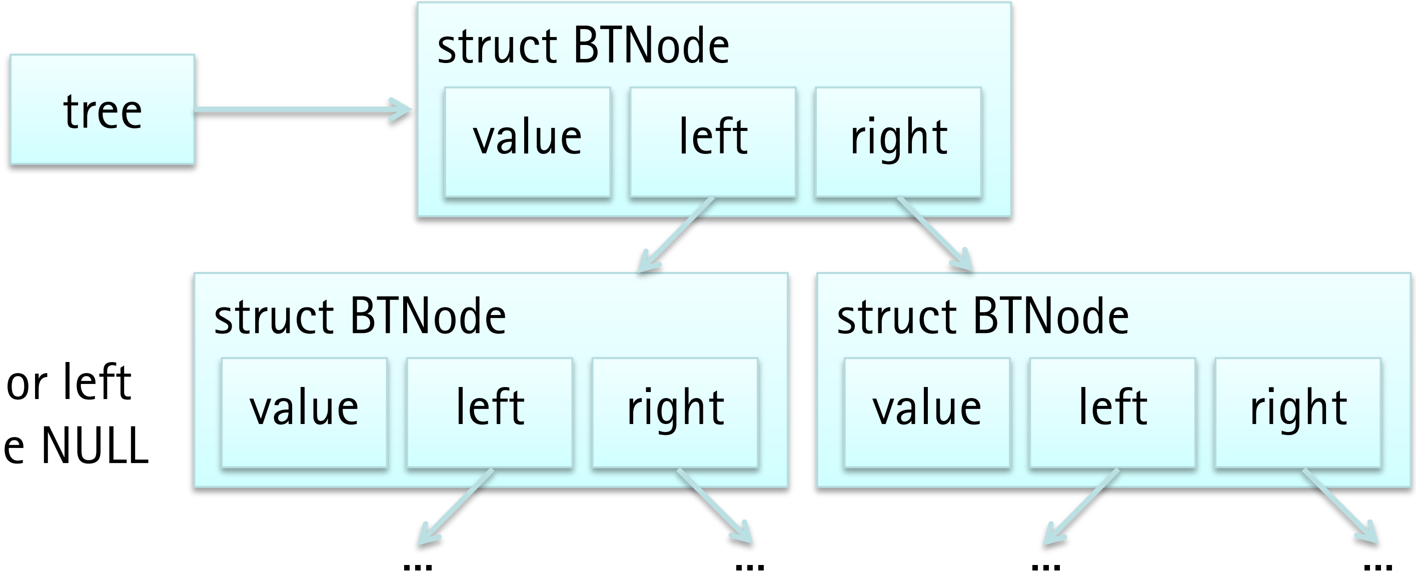
- Binary trees:

Unterschied zwischen Listen und Binärbäumen?

- Linked lists:



- Binary trees:



right or left
may be NULL

Efficient Search in Ordered Binary Trees

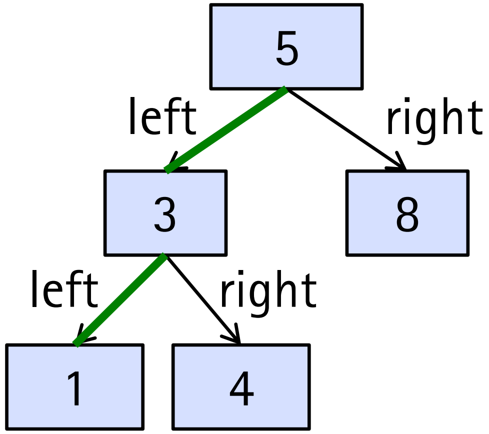
```
BTNode* search_ordered(BTNode* tree, int x) {




}
```

search_ordered(tree, 2)

Binary Tree (ordered)

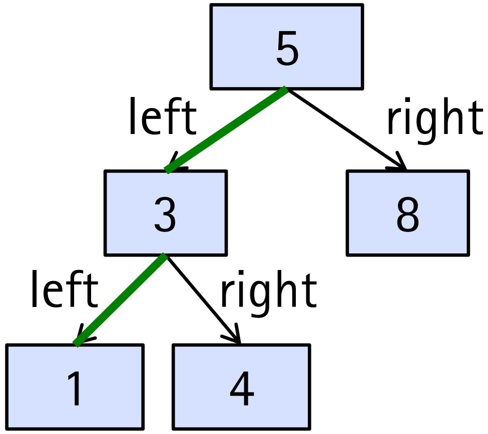


Efficient Search in Ordered Binary Trees

```
BTNode* search_ordered(BTNode* tree, int x) {
    if (tree == NULL) return NULL;
    if (x == tree->value) return tree;
    if (x < tree->value) return search_ordered(tree->left, x);
    return search_ordered(tree->right, x);
}
```

Binary Tree (ordered)

search_ordered(tree, 2)



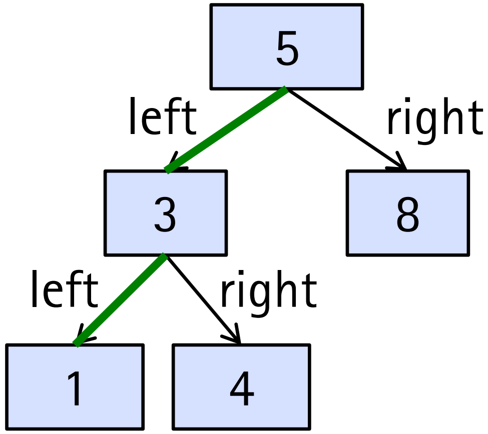
Efficient Search in Ordered Binary Trees

```
BTNode* search_ordered(BTNode* tree, int x) {
    if (tree == NULL) return NULL;
    if (x == tree->value) return tree;
    if (x < tree->value) return search_ordered(tree->left, x);
    return search_ordered(tree->right, x);
}
```

geht das auch iterativ?

Binary Tree (ordered)

search_ordered(tree, 2)



Efficient Search in Ordered Binary Trees – Iterative Version

```
BTNode* search_ordered_tree_iter(BTNode* tree, int x) {
    while (tree != NULL) {




    }
    return NULL;
}
```

Efficient Search in Ordered Binary Trees – Iterative Version

```
BTNode* search_ordered_tree_iter(BTNode* tree, int x) {
    while (tree != NULL) {
        if (x < tree->value) tree = tree->left;
        else if (x > tree->value) tree = tree->right;
        else return tree;
    }
    return NULL;
}
```