Block 5 (Dienstag 20.2.2024)

VIDEO: `video04e_backtracking_r`

6.5 Backtracking

Basic idea: if you cannot calculate directly a solution: try systematically (all) different possible solutions. HOW?

Backtracking: Take one decision (assignment of variables etc) after the other. If already taken decisions prevent a solution → withdraw from some decisions in a systematic way and try other possibilities.

Example: N Queens Problem

N queens shall be placed on a $N \times N$ chess board such that no queens checks against any other queen. This means that in each column, each row and each diagonal at most one queen is placed.

□

[Activator]

Please think for yourself for 4 minutes about a suitable algorithm (basic idea) which solves the N Queens Problem.
Next, discuss for 3 minutes with your bench neighbor about your solutions.

ATTENTION: Do not read beyond this point before you thought about the algorithm.

Basic idea of the approach: put on each column `c` in a systematic way one queen at position (row) (`pos[c]`, $c = 0, \ldots, N-1$). If this does not lead to a solution, then *backtrack*.
Additional variables for occupation of rows and diagonals. This need additional memory, which is in theory redundant but makes test faster whether rows or diagonals are available.
Since $x, y = 0, \ldots, N-1 \rightarrow$ downward diagonals: $x + y \in 0, \ldots, 2N-2$, upward diagonals: $x - y \in -N+1, \ldots, N-1$ (for C Array: add $N-1$ to start at 0)

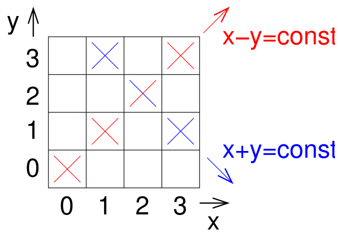


Figure 4: Test variables, whether there are queens placed in the diagonals.

```
void queens(int c, int N, int *pos, int *row,
            int *diag_up, int *diag_down)
{
  int r, c2;                                        /* loop counters */
  if(c == -1)                                    /* solution found ? */
  {
      /* omitted here */                              /* print solution */
  }
  for(r=N-1; r>=0; r--)   /* place queen in all rows of column c */
  {
    if(!row[r]&&!diag_up[c-r+(N-1)]&&!diag_down[c+r]) /*  place ?*/
    {
      row[r] = 1; diag_up[c-r+(N-1)] = 1; diag_down[c+r] = 1;
      pos[c] = r;
      queens(c-1, N, pos, row, diag_up, diag_down);
      row[r] = 0; diag_up[c-r+(N-1)] = 0; diag_down[c+r] = 0;
    }
  }
  pos[c] = 0;
}
```

Initially: `pos[i]=row[i]=diag_down[i]=diag_up[i]=0` for all `i` and call : `queens(N-1,N,pos,row,diag_up,diag_down)`.

[Activator]

Solve the 4×4 problem. How many solutions do exist?

If one counts for small values of N the number of solutions $\rightarrow$ grows exponentially as function of N.

VIDEO: `video05a_lists_r`

7 Advanced Data Structures

To perform elementary operations (store, search, read out and delete of data).

By using sophisticated data structures: faster simulations, thus, usually larger systems can be treated.

7.1 Lists

Lists = generalizations of arrays: have also linear order but more flexible
Example: Removal of array elements $O(N)$, (linked) lists: $O(1)$.

Here: single-connected lists. Data structure:

```
/* data structures for list elements */
struct elem_struct
{
    int                 info;               /* holds ‘‘information’’ */
    struct elem_struct *next;  /* points to successor (NULL if last) */
};

typedef struct elem_struct elem_t;   /* define new type for elements */
```

Double-linked lists: have also an entry `struct elem_struc *prev`
Generation and deletion of elements:

```
/****************** create_element() ****************/
/** Creates an list element an initialized info      **/
/** PARAMETERS: (*)= return-paramter                 **/
/**        value: of info                            **/
/** RETURNS:                                         **/
/**     pointer to new element                       **/
/****************************************************/
elem_t *create_element(int value)
{
  elem_t *elem;

  elem = (elem_t *) malloc (sizeof(elem_t));
  elem->info = value;
  elem->next = NULL;
  return(elem);
}

/****************** delete_element() ****************/
/** Deletes a single list element (i.e. only if it   **/
/** is not linked to another element)               **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         elem: pointer to element                 **/
/** RETURNS:                                         **/
/**    0: OK, 1: error                               **/
/****************************************************/
int delete_element(elem_t *elem)
{
  if(elem == NULL)
  {
    fprintf(stderr, "attempt to delete ‘nothing‘\n");
    return(1);
  }
  else if(elem->next != NULL)
  {
    fprintf(stderr, "attempt to delete linked element!\n");
    return(1);
  }
  free(elem);
  return(0);
}
```

Actual access to list = pointer to first element:

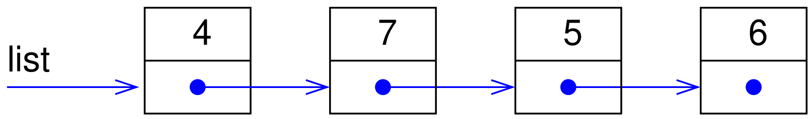


Figure 5: A single-linked list.

Generation of lists: insert elements, one after the other, either a) at beginning or b) after an existing element:

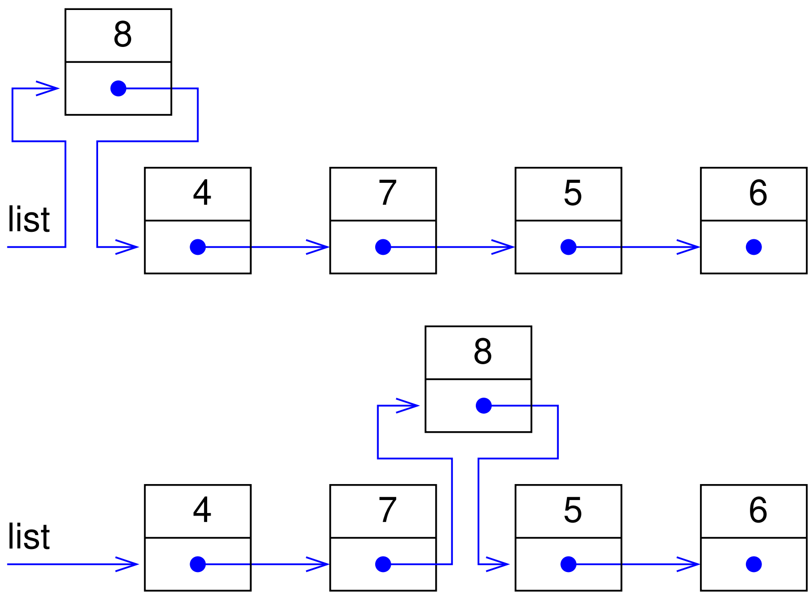


Figure 6: Insert an element into the list.

```
/****************** insert_element() ****************/
/** Inserts the element ‘elem‘ in the ‘list         **/
/** BEHIND the ‘where‘. If ’where’ is equal to NULL  **/
/** then the element is inserted at the beginning of **/
/** the list.                                        **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         list: first element of list              **/
/**         elem: pointer to element to be inserted  **/
/**        where: position of new element            **/
/** RETURNS:                                         **/
/**  (new) pointer to the beginning of the list      **/
/****************************************************/
elem_t *insert_element(elem_t *list, elem_t *elem, elem_t *where)
{
  if(where==NULL)                     /* insert at beginning ? */
  {
    elem->next = list;
    list = elem;
  }
  else                                     /* insert elsewhere */
  {
    elem->next = where->next;
    where->next = elem;
  }
  return(list);
}
```

Print a list: iterate through all elements:

```
/****************** print_list() ****************/
/** Prints all elements of a list              **/
/** PARAMETERS: (*)= return-paramter           **/
/**           list: first element of list      **/
/** RETURNS:                                   **/
/**          nothing                           **/
/************************************************/
void print_list(elem_t *list)
{
  while(list != NULL)               /* run through list */
  {
    printf("%d ", list->info);
    list = list->next;
```

```
  }
  printf("\n");
}
```

[Activator]

Write a function `elem_t *list_last(elem_t *list)`, which returns a pointer to the last element of the list.

Removal of elements: a) first element b) other elements:

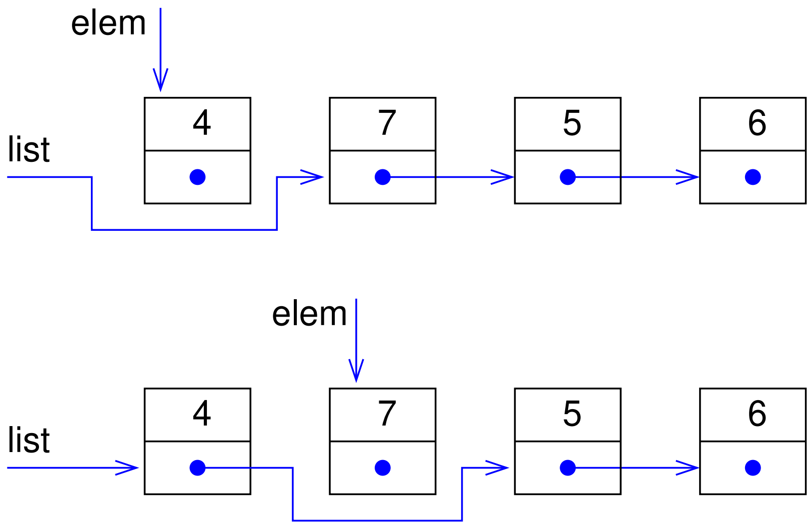


Figure 7: Removal of a list element.

One has to obtain the element <u>bevor</u> the element which is to be removed. Simpler: double-linked lists.

VIDEO: `video05b_tree1_r` VIDEO: `video05c_tree2_r`

7.2 Binäry Search Trees

Search operations for lists: $O(N)$

Better: binary search trees:

binary trees:
each element (called node) has up to two successors.
Element without predecessor = the root of the tree
each node = root of a sub tree, consisting of the node + all direct and indirect successors.

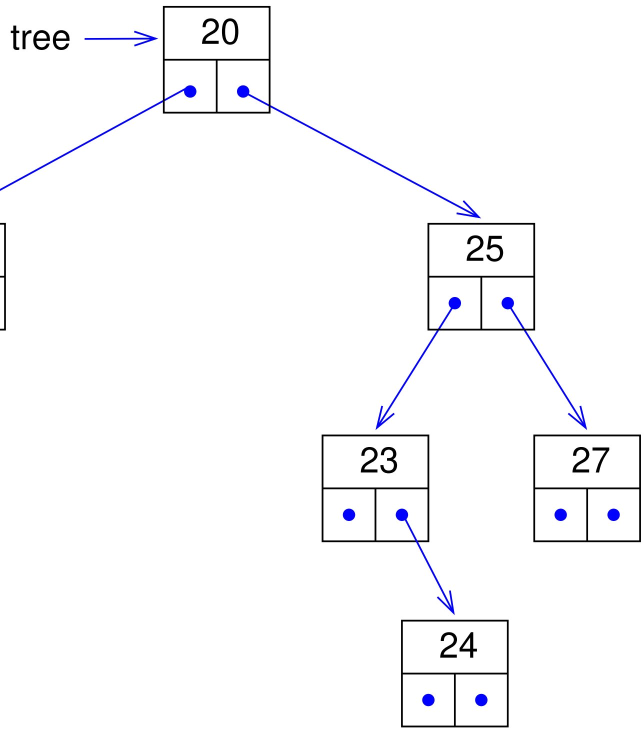


Figure 8: Binary search tree.

Search tree:
left sub tree: elements are “smaller” than root
right sub tree: elements are “larger” than root
holds also for all sub trees
$\rightarrow$ search an element: compare with root. Either element is found, or search in the left (element smaller than root) or right sub tree (repeated within a loop). If at one point sub tree does not exist: element is not contained in tree.
$\rightarrow$ search runs in $O(\log N)$ (typically).
Access to tree: pointer to root
nodes without successor = *leaves*

Basic data structure

```
/* data structures for tree elements */
struct node_struct
{
    int                  info;                  /* holds ``information'' */
    struct node_struct  *left;  /* points to left subtree (NULL if none) */
    struct node_struct *right; /* points to right subtree (NULL if none) */
};

typedef struct node_struct node_t;          /* define new type for nodes */
```

Creation and deletion of nodes: like for lists.

[Activator]

How could an algorithm for inserting an element look like? Think for yourself for 3 minutes, then discuss with your bench neighbor.

ATTENTION: Do not read beyond this point before you thought about the algorithm.

Insertion of a node:
Search for node/value
If found: stop
If not found: insert node as a leaf where the search terminated unsuccessfully.

```
/******************** insert_node() *****************/
/** Inserts 'node' into the 'tree' such that the    **/
/** increasing order is preserved                   **/
/** if node exists already, nothing happens         **/
/** PARAMETERS: (*)= return-paramter                **/
/**         tree: pointer to root of tree           **/
/**         node: pointer to node                   **/
/** RETURNS:                                        **/
/**   (new) pointer to root of tree                 **/
/****************************************************/
node_t *insert_node(node_t *tree, node_t *node)
{
  node_t *current;

  if(tree==NULL)
    return(node);
```

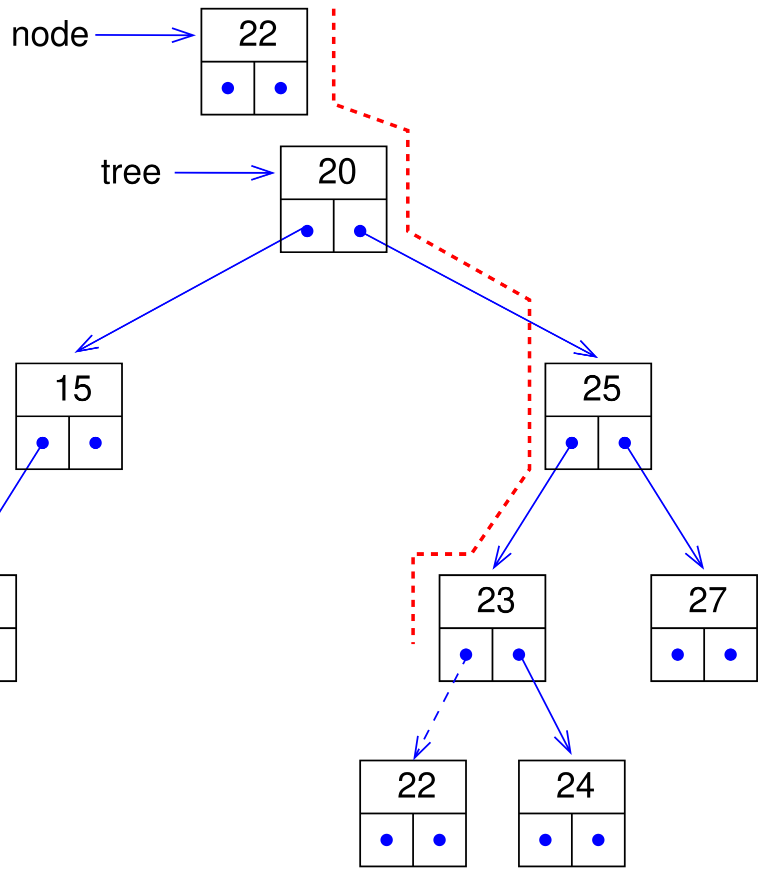


Figure 9: Inserting a new element into a search tree

```
current = tree;
while( current != NULL)                        /* run through tree */
{
  if(current->info==node->info)  /* node already contained ? */
    return(tree);
  if( node->info < current->info)                  /* left subtree */
  {
    if(current->left == NULL)
    {
      current->left = node;                            /* add node */
      return(tree);
    }
    else
      current = current->left;            /* continue searching */
  }
  else                                           /* right subtree */
```

```
    {
      if(current->right == NULL)
      {
        current->right = node;                    /* add node */
        return(tree);
      }
      else
        current = current->right;         /* continue searching */
    }
  }
}
```

[Activator]

How can one print a tree? Think for yourself for three minutes, then discuss in groups of two or three.

ATTENTION: Do not read beyond this point before you thought about the algorithm.

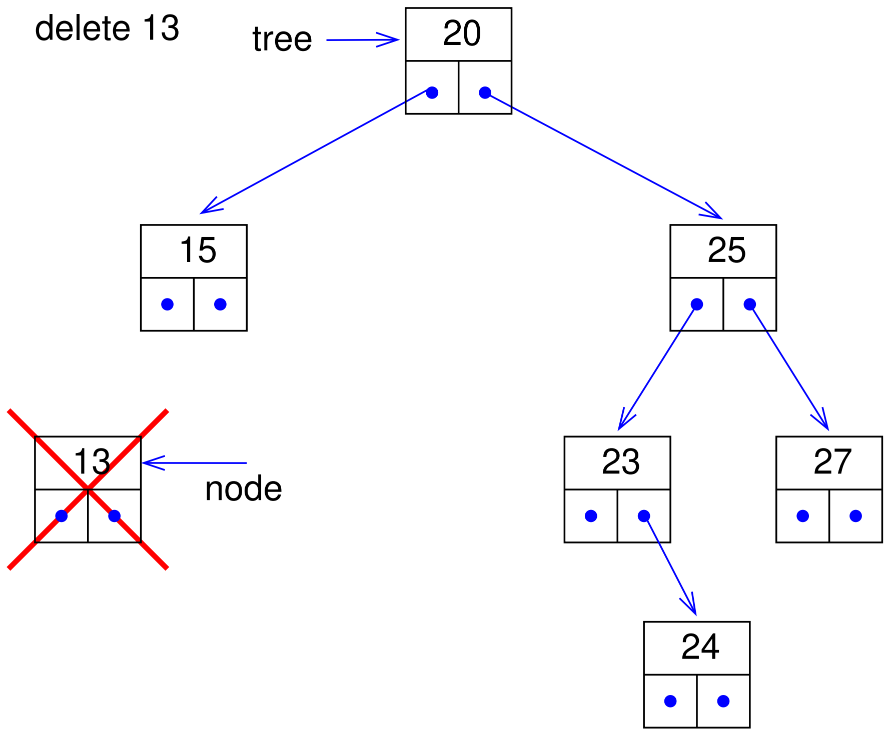
Ordered output of a tree: recursively

```
/******************** print_tree() ******************/
/** Prints tree in ascending order recursively.     **/
/** PARAMETERS: (*)= return-paramter                **/
/**         tree: pointer to root of tree           **/
/** RETURNS:                                        **/
/**   nothing                                       **/
/****************************************************/
void print_tree(node_t *tree)
{
  if(tree != NULL)
  {
    print_tree(tree->left);
    printf("%d ", tree->info);
    print_tree(tree->right);
  }
}
```

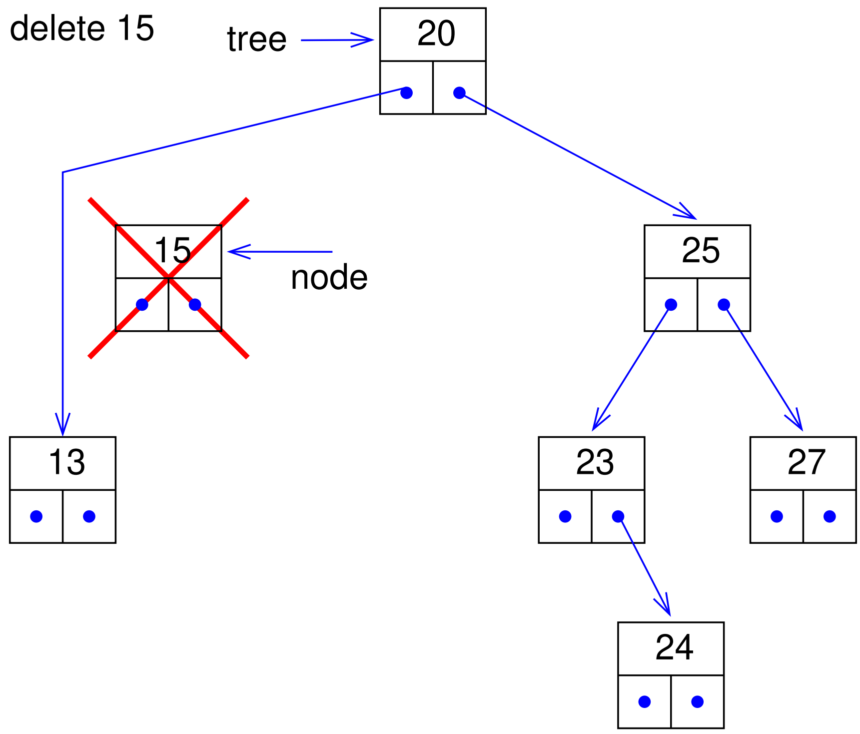
Remark: Also preorder (First print node, then left sub tree, then right sub tree) and postorder ... are possible.

Removal of a node with value x. Three cases:

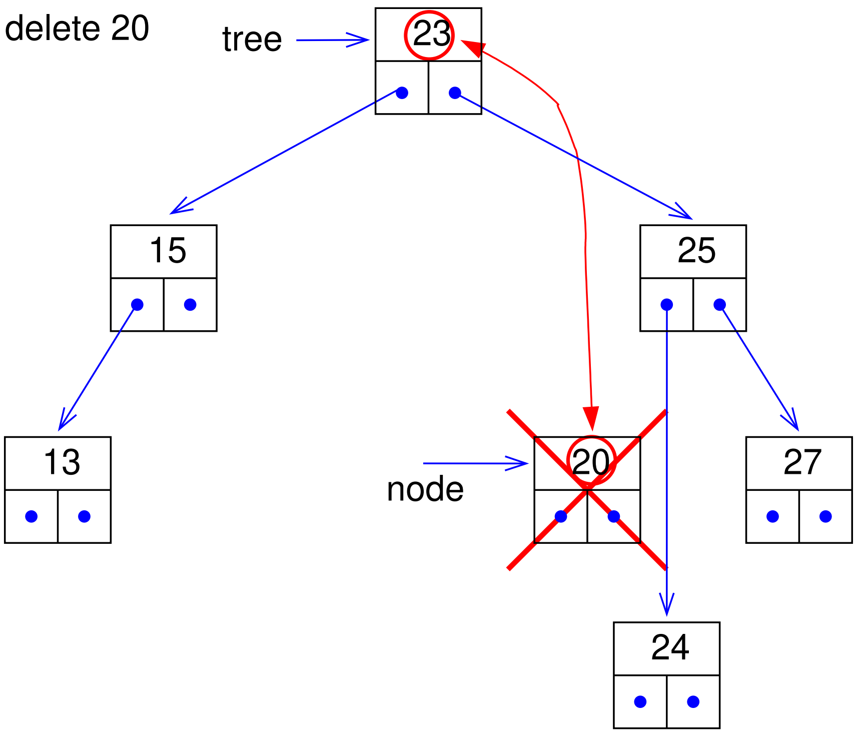
- Value is contained in a leaf (no successor): Simply remove.



- Node with x has one successor: link from predecessor to successor, omitting the node



- Node with x has two successors:
  Search for node n_2 which contains the smallest value y in the right sub tree (i.e. n_2 has <u>no</u> left successor).
  Exchange the values x and y. Now remove n_2 (which now contains x and has at most one successor) as in cases one or two .



[Activator]

Write a *recursive* function `int tree_size(node_t *tree)` which calculates the number of nodes in a tree.

Remark: Binary trees can be <u>unbalanced</u> (also called <u>degenerate</u>) (if two sub tree differ in height by more than one level)
Example: Iterated input to `insert_node` is ordered → tree becomes actually a list, thus, search takes $O(N)$ steps (worst case).
Solution: <u>Balanced trees</u> (e.g.. "red-black trees"): If a tree is unbalanced, it is balanced (e.g.. by "rotations") → all operations take only $O(\log N)$.