Block 5 (Dienstag 20.2.2024)

VIDEO: `video04e_backtracking_r`

6.5 Backtracking

Grundidee: Wenn man Lösung nicht direkt berechnen kann: versuche (systematisch) verschiedene Möglichkeiten.

Backtracking: Zuvor gemachte Entscheidungen verhindern die Lösung → nehme Entscheidungen in systematischer Weise zurück und versuche anderen Weg.

Example: N Damen Problem

N Damen sind auf einem $N \times N$ Schachbrett so zu platzieren, dass keine Dame eine andere bedroht.

Das bedeutet, dass in jeder Reihe, jeder Spalte und jeder Diagnonale maximal eine Dame steht. □

[Selbsttest]

Überlegen Sie sich erst für 4 Minuten einen Algorithmus (Grundidee), der das N Damen Problem löst.
Dann diskutieren Sie 3 Minuten mit ihrem Nachbarn ihre Lösungen.

ACHTUNG: Lesen Sie den Rest des Abschnitts NICHT, bevor Sie sich etwas überlegt haben

---

Grundidee des Verfahrens: stelle in jede Spalte `c` in systematischer Weise eine Dame auf (`pos[c]`, $c = 0, \ldots, N-1$). Wenn es keine Lösung gibt, *backtracke*.
Zusätzlich Variable für die Reihen und die Diagonalen (mehr Speicher nötig) → Test wird schneller.
Da $x, y = 0, \ldots, N-1$ → Abwärtsdiagonalen: $x + y \in 0, \ldots, 2N-2$, Aufwärtsdiagonalen: $x - y \in -N+1, \ldots, N-1$ (für C Array: immer $N-1$ addieren)

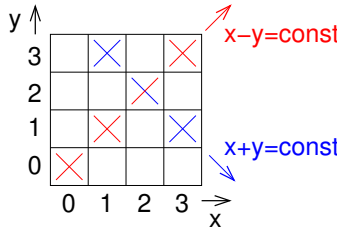


Figure 4: Testvariablen, ob in den jeweiligen Diagonalen Damen stehen.

```
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;
}
```

Anfänglich: `pos[i]=row[i]=diag_down[i]=diag_up[i]=0` für alle `i` und rufe auf:
`queens(N-1,N,pos,row,diag_up,diag_down)`.

[Selbsttest]

Lösen Sie das 4×4 Problem mit Hilfe des ausgeteilten Musters. Wieviele verschiedene Lösungen gibt es?

Wenn man für kleine N die Zahl der Lösungen zählt $\rightarrow$ wächst exponentiell mit N.

VIDEO: `video05a_lists_r`

7 Fortgeschrittene Datentypen

Zum Durchführen von Elementaren Operationen (Speichern, Suchen, Auslesen und Löschen von Daten).

Mittels guten Datenstrukturen: fast immer größere Systeme oder schneller Simulationen möglich.

7.1 Listen

Listen = Verallgemeinerungen von Arrays: auch lineare Ordnung aber flexibler. Bsp.: Löschen eines Array Elements $O(N)$, (linked) Liste: $O(1)$.
Hier: einfach verbundene Liste. Datenstruktur:

```
/* 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 */
```

Doppelt verbundene Listen: auch Eintrag `struct elem_struc *prev`
Erzeugen und Löschen von Elementen:

```
/****************** 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);
}
```

Liste = Pointer auf erstes Element:

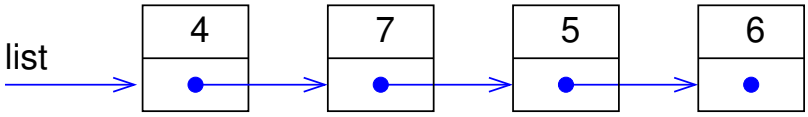


Figure 5: A single-linked list.

Erzeugung von Listen: Einfügen von Elementen a) am Anfang b) nach einem anderen Element:

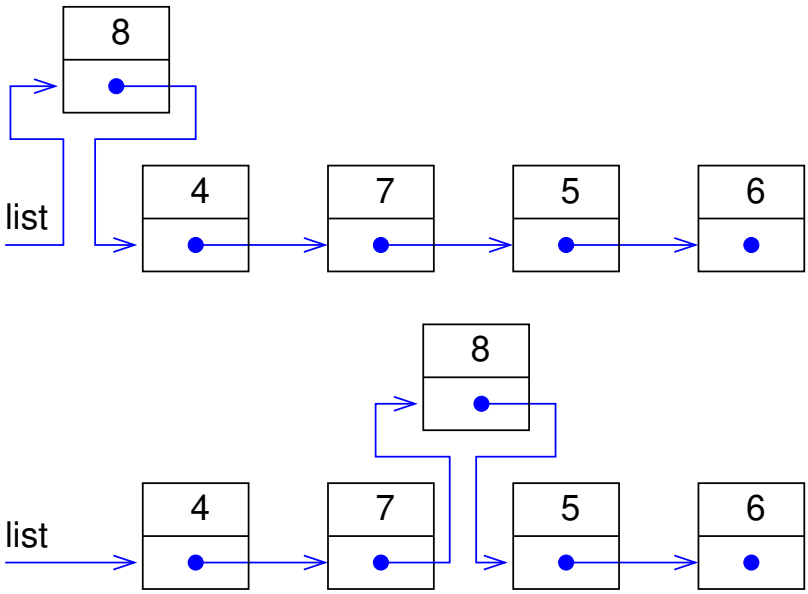


Figure 6: Einsetzen eines Elements in Liste.

```
/****************** 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);
}
```

Ausgeben einer Liste: Gehe durch alle Elemente:

```
/****************** 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");
}
```

[Selbsttest]

Schreiben Sie eine Funktion `elem_t *list_last(elem_t *list)`, die einen Pointer auf das letzte Listenelement zurück gibt.

Löschen von Elementen: a) erstes Element b) andere Elemente:

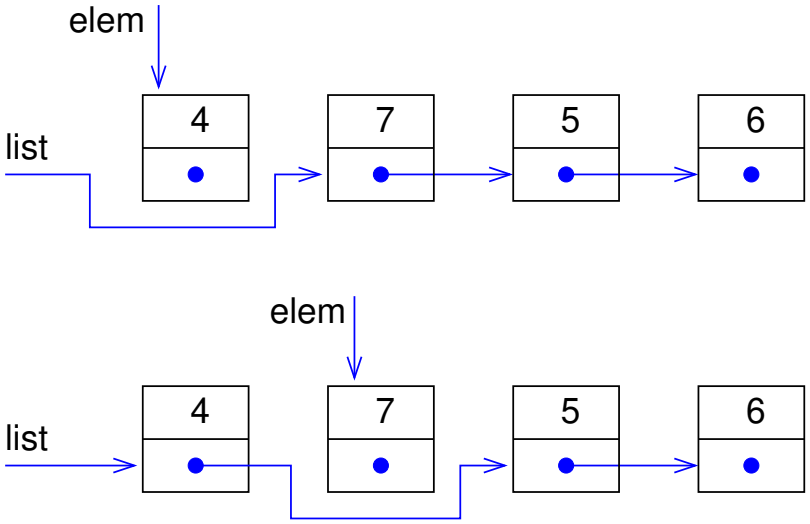


Figure 7: Löschen eines Listenelements

Man muss das Element vor dem zu löschenden Element finden. Einfacher: doppelt verkettete Listen.

VIDEO: `video05b_tree1_r` | VIDEO: `video05c_tree2_r`

7.2 Binäre Suchbäume

Suchoperation für Listen: $O(N)$

Besser: binäre Suchbäume:

binäre Bäume:
Jedes Element (genannt Knoten) hat bis zu zwei Nachfolger.
Element ohne Vorgänger = Wurzel des Baumes
Jeder Knoten = Wurzel des Unterbaums, der Knoten + alle seine direkten + indirekten Nachfolger enthält.

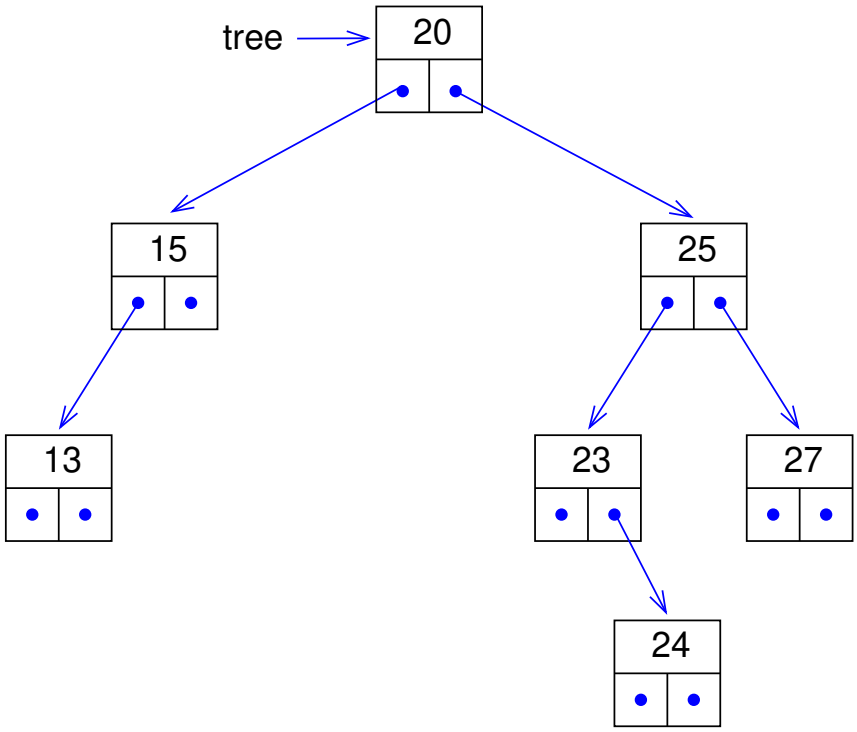


Figure 8: Binärer Suchbaum.

Suchbaum:
Linker Unterbaum: Elemente sind "kleiner" als an Wurzel
Rechter Unterbaum: Elemente sind "größer" als an Wurzel
gilt auch für jeden Unterbaum
→ Suche nach Element: vergleiche mit Wurzel, entweder gefunden, oder im linken (Element kleiner als Wurzel) oder rechten U-Baum (mit Schleife) weitersuchen. Falls U-Baum nicht existiert: Element nicht enthalten.
→ Suche funktioniert in $O(\log N)$ (typischerweise).
Baum: repräsentiert durch Zeiger auf Wurzel
Knoten ohne Nachfolger = Blatt

Grundlegende Datenstruktur

```
/* 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 */
```

Erzeugen und Löschen von einzelnen Knoten: analog zu Listen.

[Selbsttest]

Wie könnte ein Algorithmus zum Einsetzen eines Elements aussehen. Überlegen Sie erst 3 Minuten selber, dann diskutieren Sie mit Ihrem Nachbarn.

ACHTUNG: Lesen Sie den Rest des Abschnitts NICHT, bevor Sie sich etwas überlegt haben

Einsetzen eines Knotens:
Suche nach Wert
Falls gefunden: Ende
Falls nicht: Setze Werte als Blatt dort ein, wo Suche endete

```
/********************* 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);
  current = tree;
```

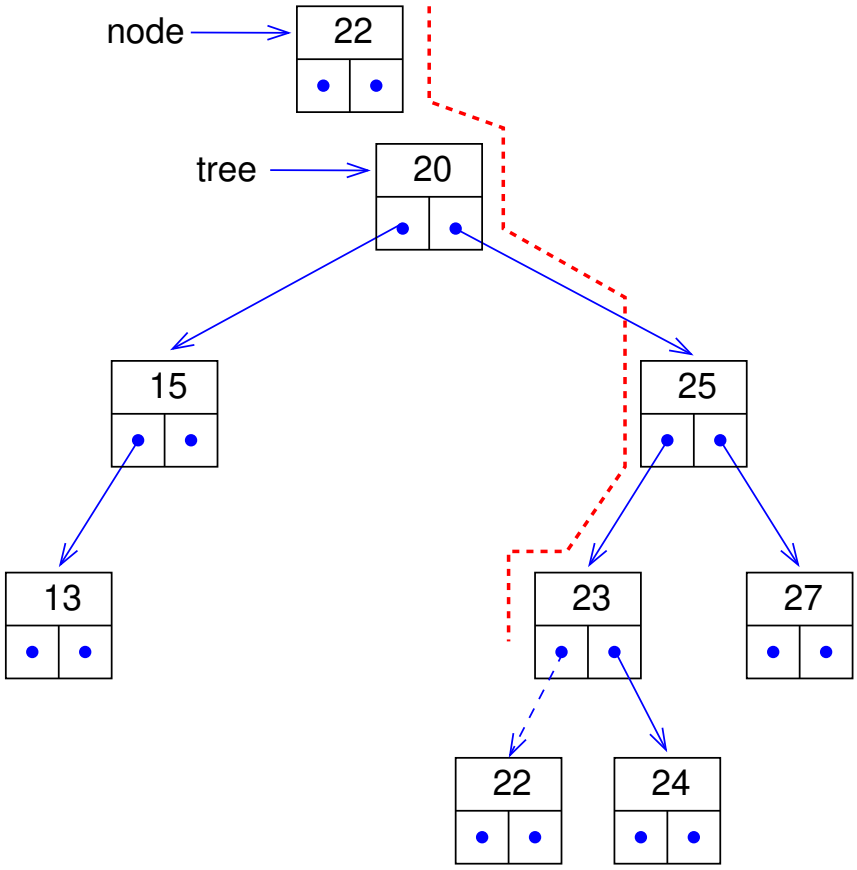


Figure 9: Einsetzen eines neuen Elementes in den Suchbaum

```
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 */
    }
  }
}
```

[Selbsttest]

Wie kann man einen Baum ausgeben? Überlegen Sie drei Minuten und diskutieren dann drei Minuten zu zweit/dritt.

ACHTUNG: Lesen Sie den Rest des Abschnitts NICHT, bevor Sie sich etwas überlegt haben

---

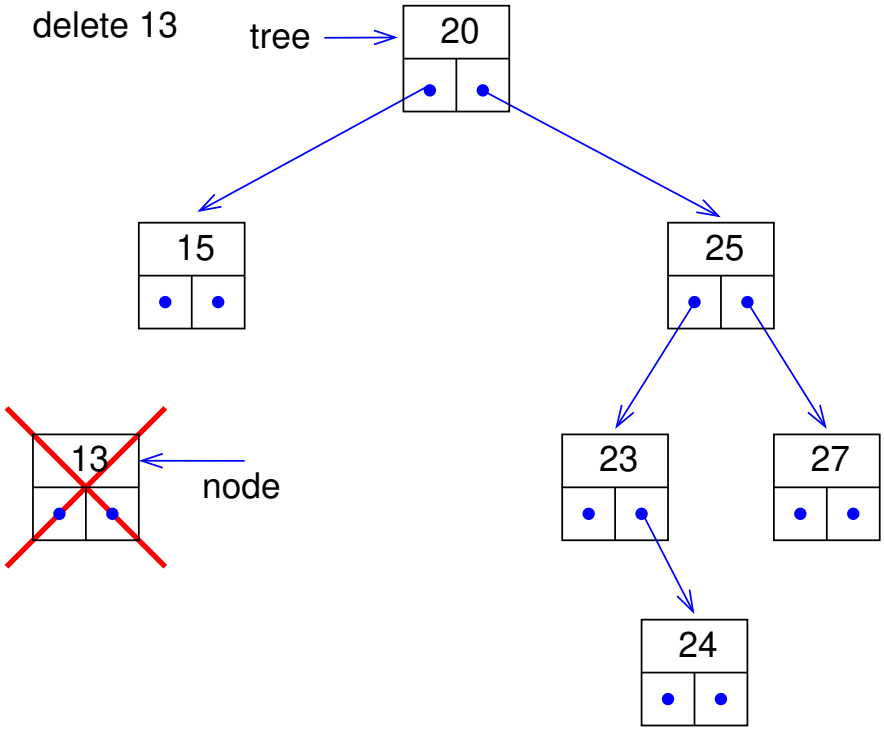
Geordnete Ausgabe des Baums: rekursiv

```
/********************* 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);
  }
}
```

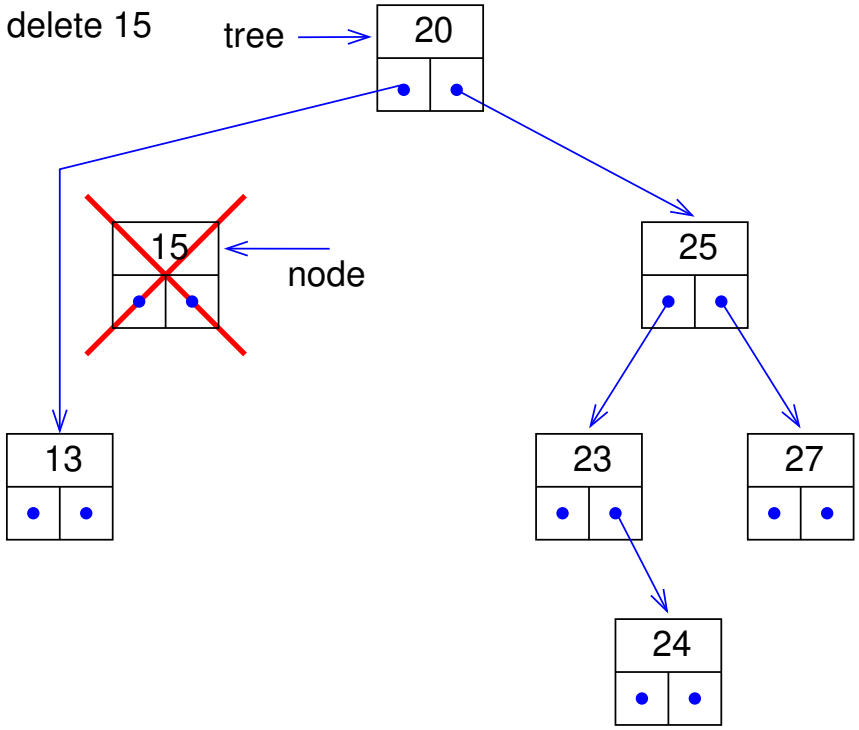
Hinweis: Auch preorder (Erst Knoten, dann linken, dann rechten Unterbaum ausgeben) und postorder möglich.

Entfernen eines Wertes x. 3 Fälle:

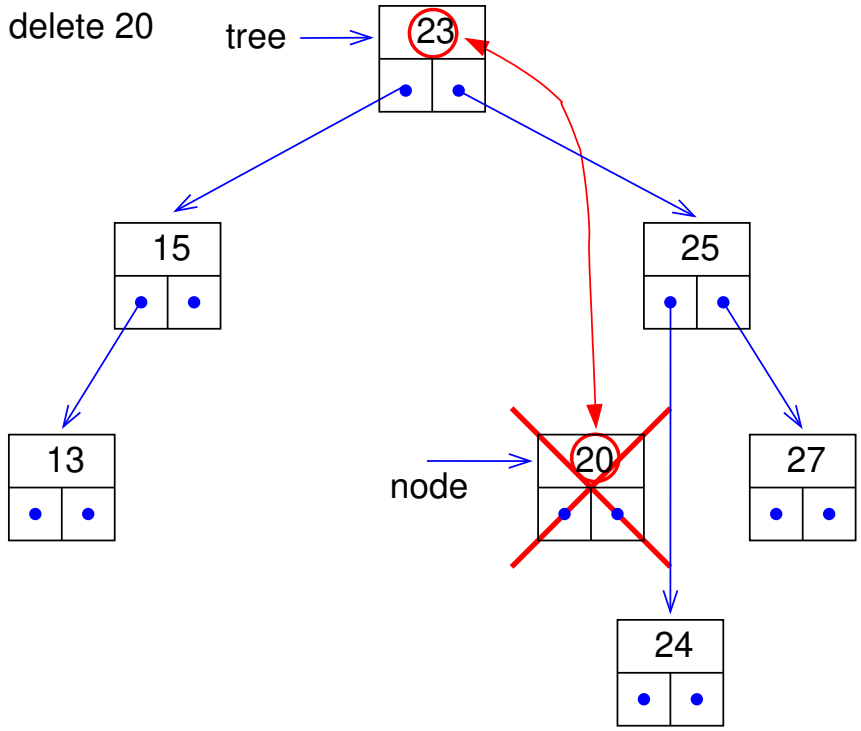
- Wert ist in Blatt (keine Nachfolger): Einfach entfernen.



- Knoten mit x hat einen Nachfolger: Knoten durch Nachfolger ersetzen



- Knoten mit x hat zwei Nachfolger:
  Such nach Knoten n_2, der den kleinsten Wert y im rechten Unterbaum hat (d.h. n_2 hat keinen linken Nachfolger).
  Vertausche Werte x und y. Entferne n_2 wie in Fällen eins oder zwei.



[Selbsttest]

Schreiben Sie eine *rekursive* Funktion `int tree_size(node_t *tree)` die Zahl der Knoten im Baum berechnet.

Hinweis: Binäre Bäume können unbalanciert (oder entartet) sein (wenn zwei Unterbäume sich um mehr als ein in der Höhe unterschieden)
Bsp: Iterierte Eingabe zu `insert_node` ist geordnet → Baum wird zur Liste, also Suche dauert $O(N)$ (worst case).
Lösung: Balancierte Bäume (z.B. "rot-schwarz Bäume"): Falls Baum unbalanciert ist, wird er ausgeglichen (z.b. durch "Rotationen") → Alle Operationen dauern nur $O(\log N)$.