Listen, Keller, Schlangen

Einfach verkettete Listen

Eine Zeiger-Implementierung von einfach-verketteten Listen, also Listen mit Vorwärtszeigern.

```
//Deklarationsdatei liste.h fuer einfach-verkettete Listen.
enum boolean {False, True };
class liste{

private:
   typedef struct Element {
     int data;
     Element *next;};
   Element *head, *current;

public:    liste( ) // Konstruktor
    { head = new Element; current = head; head->next = 0; }
```

Public: Die weiteren Operationen

- **void insert(int data):**
  Ein neues Element mit Wert data wird nach dem gegenwärtigen Element eingefügt. Der Wert von current ist unverändert.
- **void remove( ):** Das dem gegenwärtigen Element folgende Element wird entfernt. Der Wert von current ist unverändert.
- **void movetofront( ):** current erhält den Wert head.
- **void next( ):** Das nächste Element wird aufgesucht. Dazu wird current um eine Position nach rechts bewegt.
- **boolean end( ):** Ist das Ende der Liste erreicht?
- **boolean empty( ):** Ist die Liste leer?
- **int read( ):** Gib das Feld data des nächsten Elements aus.
- **void write(int wert):** Überschreibe das Feld data des nächsten Elements mit der Zahl wert.
- **void search(int wert):** Suche, rechts von der gegenwärtigen Zelle, nach der ersten Zelle z mit Datenfeld wert. Der Zeiger current wird auf den Vorgänger von z zeigen.

Eigenschaften

- Jede Liste besitzt stets eine leere Kopfzelle: **liste()** erzeugt diese Zelle, nachfolgende Einfügungen können nur nach der Kopfzelle durchgeführt werden.
- Alle Operationen –bis auf search– werden in konstanter Zeit $O(1)$ unterstützt. Die search-Funktion benötigt möglicherweiser Zeit proportional zur Länge der Liste.

The good and the bad

+ Die Größe der Liste ist proportional zur Anzahl der gespeicherten Elemente: Die Liste passt sich der darzustellenden Menge an.
- Die Suche dauert viel zu lange.

Die Addition dünn besetzter Matrizen

A und B sind Matrizen mit n Zeilen und m Spalten.
Berechne die Summe $C = A + B$.

- Das Programm

```
for (i=0 ; i < n; i++)
  for (j=0 ; j < m; j++)
    C[i,j] = A[i,j] + B[i,j];
```

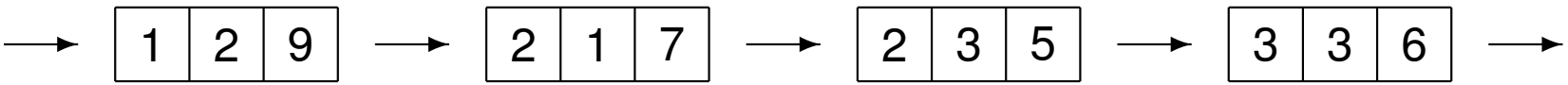
  bestimmt C in Zeit $O(n \cdot m)$.

- **Ziel**: Bestimme C in Zeit $O(a + b)$, wobei a und b die Anzahl der von Null verschiedenen Einträge von A und B ist.

Listendarstellung von Matrizen

- Stelle A und B durch einfach verkettete Listen L_A und L_B in Zeilenordnung dar.
- Jedes Listenelement speichert neben dem Wert eines Eintrags auch die Zeile und die Spalte des Eintrags.

Die Zeilenordnung für $A \equiv \begin{bmatrix} 0 & 9 & 0 \\ 7 & 0 & 5 \\ 0 & 0 & 6 \end{bmatrix}$ ist



Der Algorithmus

(1) Beginne jeweils am Anfang der Listen L_A und L_B.

(2) Solange beide Listen nicht-leer sind, wiederhole

  (a) das gegenwärtige Listenelement von L_A (bzw. L_B) habe die Koordinaten (i_A, j_A) (bzw. (i_B, j_B)).

  (b) Wenn $i_A < i_B$ (bzw. $i_A > i_B$), dann füge das gegenwärtige Listenelement von L_A (bzw. L_B) in die Liste L_C ein und gehe zum nächsten Listenelement von L_A (bzw. L_B).

  (c) Wenn $i_A = i_B$ und $j_A < j_B$ (bzw. $j_A > j_B$), dann füge das gegenwärtige Listenelement von L_A (bzw. L_B) in die Liste L_C ein und gehe zum nächsten Listenelement von L_A (bzw. L_B).

  (d) Wenn $i_A = i_B$ und $j_A = j_B$, dann addiere die beiden Einträge und füge die Summe in die Liste L_C ein. Die Zeiger in beiden Listen werden nach rechts bewegt.

(3) Wenn die Liste L_A (bzw. L_B) leer ist, kann der Rest der Liste L_B (bzw. L_A) an die Liste L_C angehängt werden.

Deques, Stacks und Queues

- Der abstrakte Datentyp **Stack** unterstützt die Operationen
  - `pop` : Entferne den jüngsten Schlüssel.
  - `push` : Füge einen Schlüssel ein.

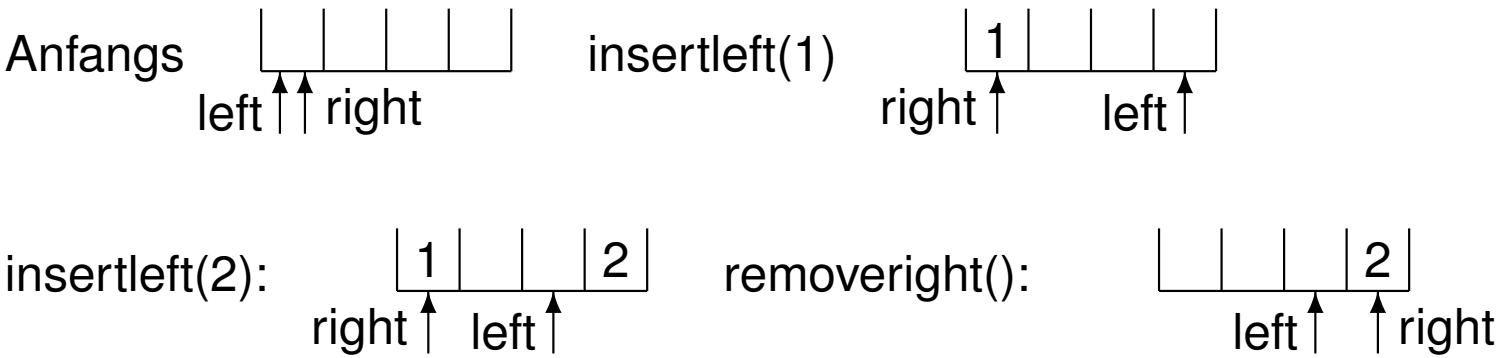
  Ein wichtiges Anwendungsgebiet ist die nicht-rekursive Implementierung rekursiver Programme.
- Eine **Queue** modelliert das Konzept einer Warteschlange und unterstützt die Operationen
  - `dequeue`: Entferne den ältesten Schlüssel.
  - `enqueue` : Füge einen Schlüssel ein.
- Stacks und Queues werden von einem **Deque** verallgemeinert. Die folgenden Operationen werden unterstützt:
  - `insertleft` : Füge einen Schlüssel am linken Ende ein.
  - `insertright` : Füge einen Schlüssel am rechten Ende ein.
  - `removeleft` : Entferne den Schlüssel am linken Ende.
  - `removeright` : Entferne den Schlüssel am rechten Ende.

Wie werden Deques und Listen implementiert?

Die Implementierung eines Deque

- Benutze ein an beiden Enden „zusammengeschweißtes“ 1-dimensionales Array.
  - Der Inhalt des Deques wird jetzt entsprechend den Operationen „über den entstandenen Ring“ geschoben.
- Benutze zwei Positionsvariable **left** und **right**: **left** steht jeweils **vor** dem linken Ende und **right** jeweils **auf** dem rechten Ende.
- **Forderung**:

  Die Zelle mit Index **left** ist stets leer und anfänglich ist **left = right**.



Eine Implementierung von Listen durch Arrays (1/2)

Eine einfach verkettete Liste bestehe aus Zellen mit einem **integer-Datenfeld** und einem **Vorwärtszeiger**.
Zu keinem Zeitpunkt möge die Liste mehr als n Elemente besitzen.

- **Daten** und **Zeiger** seien Arrays der Größe n.
- Wenn ein Listenelement den „Index i erhält“, dann ist **Daten**$[i]$ der Wert des Listenelements und **Zeiger**$[i]$ der Index des rechten Nachbarn.
- Für die Speicherverwaltung benutze eine zusätzliche Datenstruktur **Frei**, die die Menge der freien Indizes verwaltet.

  Zu Anfang ist **Frei**$= \{0, \ldots, n-1\}$.

Eine Implementierung von Listen durch Arrays (2/2)

- Wenn ein Element mit Wert w nach einem Element mit Index i einzufügen ist:

  Wenn **Frei** nicht-leer ist, dann entnimm einen freien Index j und setze **Daten**$[j] = w$, **Zeiger**$[j] =$ **Zeiger**$[i]$ und **Zeiger**$[i] = j$.

- Wenn ein Element zu entfernen ist, dessen Vorgänger den Index i besitzt:

  Füge $j =$ **Zeiger**$[i]$ in die Datenstruktur Frei ein und setze **Zeiger**$[i] =$ **Zeiger**$[j]$.

- Welche Datenstruktur ist für **Frei** zu wählen?

  Jede Datenstruktur, die es erlaubt, Elemente einzufügen und zu entfernen, tut's: ein Stack, eine Queue oder ein Deque ist OK.

Bäume

Bäume

Gewurzelte Bäume für die hierarchische Strukturierung von Daten.

- Gewurzelte Bäume als konzeptionelle Hilfsmittel für
  - von einem Benutzer angelegte Verzeichnisse,
  - als Nachfahrenbaum
  - oder für die Veranschaulichung rekursiver Programme (Rekursionsbaum oder Baum einer Rekursionsgleichung).

- Gewurzelte Bäume als Datenstrukturen von Algorithmen:
  - in der Auswertung arithmetischer Ausdrücke,
  - in der Syntaxerkennung mit Hilfe von Syntaxbäumen,
  - im systematischen Aufzählen von Lösungen (Entscheidungsbäume)
  - oder in der Lösung von Suchproblemen.

Was ist ein gewurzelter Baum?

Ein gewurzelter Baum T wird durch eine **Knotenmenge** V und eine **Kantenmenge** $E \subseteq V \times V - \{(i,i) \mid i \in V\}$ dargestellt.
Die gerichtete Kante (i,j) führt **von** i **nach** j.

Wann ist T ein gewurzelter Baum?

- T muss genau einen Knoten r besitzen, in den keine Kante hineinführt. r heißt die **Wurzel** von T.
- In jeden Knoten darf höchstens eine Kante hineinführen
- und jeder Knoten muss von der Wurzel aus erreichbar sein.

(Ab jetzt sprechen wir nur von Bäumen und meinen gewurzelte Bäume.)

Eine (knotendisjunkte) Vereinigung von Bäumen heißt ein **Wald**.

Zentrale Begriffe

- Wenn (v, w) eine Kante ist, dann nennen wir v den **Elternknoten** von w und sagen, dass w ein **Kind** von v ist.
  - Die anderen Kinder von v heißen **Geschwister** von w.
- **Aus-Grad** (v) ist die Anzahl der Kinder von v.
  - Einen Knoten b mit Aus-Grad $(b) = 0$ bezeichnen wir als **Blatt**.
  - T heißt k-**när**, falls der Aus-Grad aller Knoten höchstens k ist. Für $k = 2$ sprechen wir von **binären Bäumen**.
- Ein **Weg** von v nach w ist eine Folge $(v_0, \ldots, v_m)$ von Knoten mit $v_0 = v, v_m = w$ und $(v_i, v_{i+1}) \in E$ für alle i $(0 \leq i < m)$.
  - v ist ein **Vorfahre** von w und w ein **Nachfahre** von v.
  - Die **Länge** des Weges ist m, die Anzahl der Kanten des Weges.
  - **Tiefe**(v) ist die Länge des (eindeutigen) Weges von r nach v.
  - **Höhe**(v) ist die Länge des längsten Weges von v zu einem Blatt.
- T heißt **geordnet**, falls für jeden Knoten eine Reihenfolge der Kinder vorliegt.

Operationen auf Bäumen

(1) `Wurzel`: Bestimme die Wurzel von T.

(2) `Eltern`(v): Bestimme den Elternknoten des Knotens v in T.
Wenn $v = r$, dann ist der Null-Zeiger auszugeben.

(3) `Kinder`(v)`:` Bestimme die Kinder von v. Wenn v ein Blatt ist, dann ist der Null-Zeiger als Antwort zu geben.

(4) Für binäre geordnete Bäume:

  (4a) `LKind`(v): Bestimme das linke Kind von v.
  (4b) `RKind`(v): Bestimme das rechte Kind von v.
  (4c) Sollte das entsprechende Kind nicht existieren, ist der Null-Zeiger als Antwort zu geben.

(5) `Tiefe`(v): Bestimme die Tiefe von v.

(6) `Höhe`(v): Bestimme die Höhe von v.

(7) `Baum`($v, T_1, \ldots, T_m$): Erzeuge einen geordneten Baum mit Wurzel v und Teilbäumen $T_1, \ldots, T_m$.

(8) `Suche`(x): Bestimme alle Knoten mit Wert x.

Das Eltern-Array
(wenn nur Vorfahren interessieren)

Das Eltern-Array

Annahme: Jeder Knoten besitzt eine Zahl aus $\{1, \ldots, n\}$ als Namen und zu jedem $i \in \{1, \ldots, n\}$ gibt es genau einen Knoten mit Namen i.

- Ein integer-Array **Baum** speichert für jeden Knoten v den Namen des Elternknotens von v:

  $$\text{Baum}[v] = \text{Name des Elternknotens von } v.$$

  Wenn $v = r$, dann setze $\text{Baum}[v] = 0$.

- Das Positive:
  - \+ schnelle Bestimmung des Elternknotens (Zeit $= O(1)$)
  - \+ und schnelle Bestimmung der Tiefe von v (Zeit $= O(\text{Tiefe}(v))$).
  - \+ Minimaler Speicherplatzverbrauch:
    Bäume mit n Knoten benötigen Speicherplatz n.
- Das Negative: für die Bestimmung der Kinder muss der gesamte Baum durchsucht werden (Zeit = O(Anzahl Knoten).)

Die Binärbaum-Implementierung
(die ideale Datenstruktur für Binärbäume)

Die Binärbaum-Implementierung

Ein Knoten wird durch die Struktur

```
struct Knoten {
  int wert;
  Knoten *links, *rechts; };
```

dargestellt.

- Wenn der Zeiger z auf die Struktur des Knotens v zeigt,
  - dann ist $z \to \mathit{wert}$ der *Wert* von v und
  - $z \to \mathit{links}$ (bzw. $z \to \mathit{rechts}$) zeigt auf die Struktur des linken (bzw. rechten) Kindes von v.
  - Der Zeiger wurzel zeigt auf die Struktur der Wurzel des Baums.
- Speicherbedarf:
  - $2n$ Zeiger (zwei Zeiger pro Knoten) und
  - n Zellen (eine Zelle pro Knoten).

Die Binärbaum-Implementierung: Stärken und Schwächen

+ Die **Kinderbestimmung** gelingt schnellstmöglich, in Zeit $O(1)$.
+ **Höhe** (v, T) wird angemessen unterstützt mit der Laufzeit

$$O(\text{Anzahl der Knoten im Teilbaum mit Wurzel } v).$$

Begründung später.

- Für die **Bestimmung des Elternknotens** muss möglicherweise der gesamte Baum durchsucht werden!
- Die **Bestimmung der Tiefe** ist auch schwierig, da der Elternknoten nicht bekannt ist.

Die Kind-Geschwister Implementierung (die Allzweckwaffe)

Die Kind-Geschwister Implementierung

Ein Knoten wird durch die Struktur

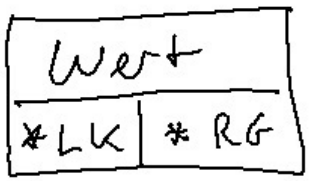
```
typedef struct Knoten {
  int wert;
  Knoten *LKind, *RGeschwister; };
```

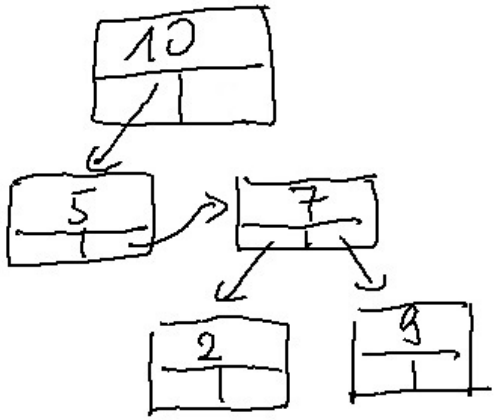
dargestellt.

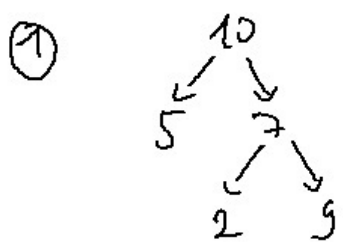
- Wenn der Zeiger z auf die Struktur des Knotens v zeigt,
  - dann ist $z \to \mathit{wert}$ der *Wert* von v und
  - $z \to$ LKind (bzw. $z \to$ RGeschwister) zeigt auf die Struktur des linken Kindes, bzw. des rechten Geschwisterknotens von v.
  - Der Zeiger wurzel zeigt wieder auf die Struktur der Wurzel des Baums.
- Im Vergleich zur Binärbaum-Darstellung:
  - Ähnliches Laufzeitverhalten und ähnliche Speichereffizienz,
  - aber jetzt lassen sich alle Bäume und nicht nur Binärbäume darstellen!

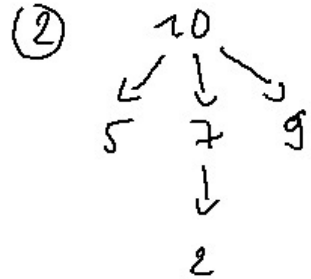
Kind-Geschwister demogr.

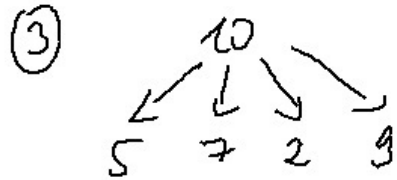
Welchem Baum entspricht die Datenstruktur?











Auflösung: (2)

Postorder, Präoder und Inorder

Suche in Bäumen: Postorder, Präorder und Inorder

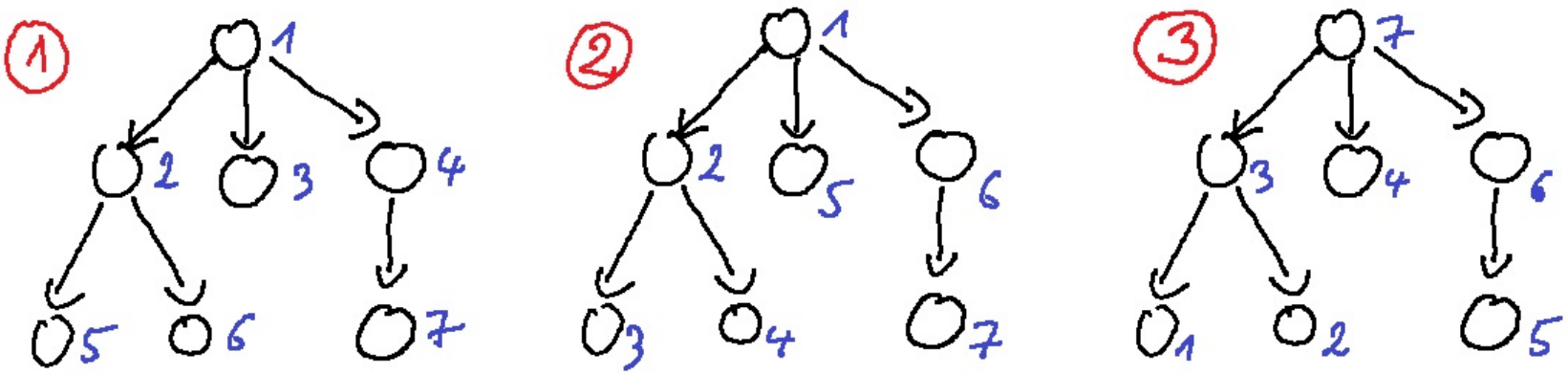
Sei T ein geordneter Baum mit Wurzel r und Teilbäumen $T_1, \ldots, T_m$.

- **Postorder**: Durchlaufe rekursiv die Teilbäume $T_1, \ldots, T_m$ nacheinander. Danach wird die Wurzel r besucht.
- **Präorder**: besuche zuerst r und danach durchlaufe rekursiv die Teilbäume $T_1, \ldots, T_m$.
- **Inorder**: Durchlaufe zuerst T_1 rekursiv, sodann die Wurzel r und letztlich die Teilbäume $T_2, \ldots, T_m$ rekursiv.

```
void praeorder (Knoten *p)
{
   if (p != nullptr)
   { cout « p→ wert;
     for ( Knoten *q=p→LKind; q != nullptr; q=q→RGeschwister)
       praeorder(q);
   }
}
```

Traversierungen demogr.

Welches ist eine Präorder-Traversierung?



Auflösung: (2)

Bestimmung der Tiefe und Höhe von Knoten

Die Struktur eines Knoten besteht aus den Feldern

tiefe, hoehe, wert, LKind und RGeschwister.

```
tiefe(wurzel,-1);
void tiefe (Knoten *p, int t)
{ t = t+1;
   if (p != nullptr)
   { p→tiefe = t;
     for (Knoten *q=p→LKind; q != nullptr; q=q→RGeschwister)
         tiefe(q,t);}}

void hoehe (Knoten *p)
{ int h=-1;
   if (p != nullptr)
   { for (Knoten *q=p→LKind; q != nullptr; q=q→RGeschwister)
     { hoehe (q); h = max ( h, q → hoehe); }
     p→hoehe = h+1; } }
```

Eine nicht-rekursive Präorder-Implementierung

Der Teilbaum mit Wurzel v ist in Präorder-Reihenfolge zu durchlaufen.

(1) Wir fügen einen Zeiger auf die Struktur von v in einen anfänglich leeren Stack ein.

(2) Solange der Stack nicht-leer ist, wiederhole:

  (a) Entferne das zuoberst liegende Stack-Element w mit Hilfe der Pop-Operation.
  /∗ w wird besucht. ∗ /

  (b) Die Kinder von w werden in umgekehrter Reihenfolge in den Stack eingefügt.
  /∗ Durch die Umkehrung der Reihenfolge werden die Bäume später in ihrer natürlichen Reihenfolge abgearbeitet. ∗ /

Die Laufzeit ist linear in der Knotenzahl n. Warum?

- Jeder Knoten wird genau einmal in den Stack eingefügt.
- Insgesamt werden also höchstens $O(n)$ Stackoperationen durchgeführt. Stackoperationen dominieren aber die Laufzeit.

Welcher Knoten wird direkt nach v besucht?

- **Postorder**:
  - Das linkeste Blatt im Baum des rechten Geschwisterknotens.
  - Wenn v keinen rechten Geschwisterknoten besitzt, dann wird der Elternknoten von v als nächster besucht.
- **Präorder**:
  - Das linkeste Kind von v, wenn v kein Blatt ist.
  - Wenn v ein Blatt ist, dann das erste nicht-besuchte Kind des tiefsten, nicht vollständig durchsuchten Vorfahren von v.

Prioritätswarteschlangen und Heaps

Prioritätswarteschlangen

Der abstrakte Datentyp „**Prioritätswarteschlange**“: Füge Elemente (mit Prioritäten) ein und entferne jeweils das Element höchster Priorität.

- Eine Schlange ist eine sehr spezielle Prioritätswarteschlange:
  - Die Priorität eines Elements richtet sich nach dem Zeitpunkt des Einfügens.
- Der abstrakte Datentyp „Prioritätswarteschlange“ umfasst die Operationen
  - **insert**(x,Priorität),
  - **delete**_max(),
  - **change_priority**(wo,Priorität*), wähle Priorität* als neue Priorität
  - und **remove**(wo), entferne das durch wo beschriebene Element.

Wir entwerfen eine geeignete Datenstruktur.

Der Heap

Ein Heap ist ein Binärbaum mit

Heap-Struktur,

der Prioritäten gemäß einer

Heap-Ordnung

abspeichert.

Ein Binärbaum T der Tiefe t hat **Heapstruktur**, wenn:

(a) jeder Knoten der Tiefe höchstens $t-2$ genau 2 Kinder hat,

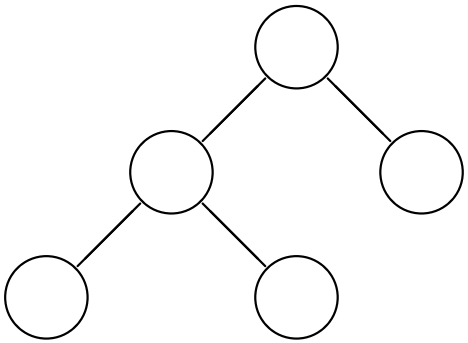
(b) für jeden Knoten v der Tiefe $t-1$ mit weniger als 2 Kindern alle Knoten der Tiefe $t-1$, die rechts von v liegen, keine Kinder haben, und

(c) falls ein Knoten v der Tiefe $t-1$ genau ein Kind hat, dieses Kind ein linkes Kind ist.

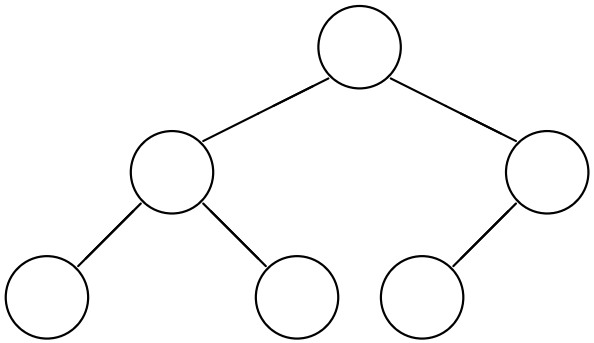
Ein Binärbaum mit Heapstruktur ist ein fast vollständiger binärer Baum: Ist v ein Knoten mit nur einem Kind, so haben alle Knoten links von v zwei Kinder, und alle Knoten rechts von v haben keine Kinder.

Heapstruktur multiple choice

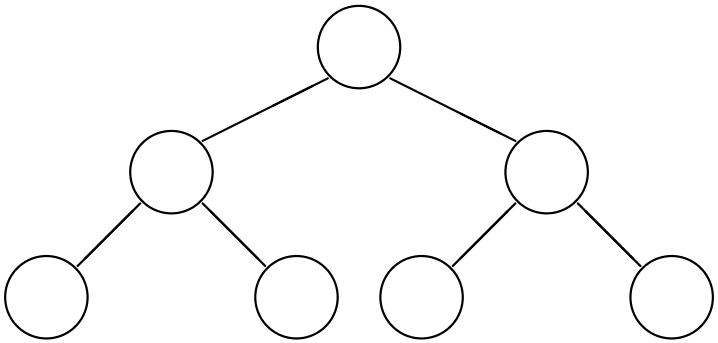
(1)



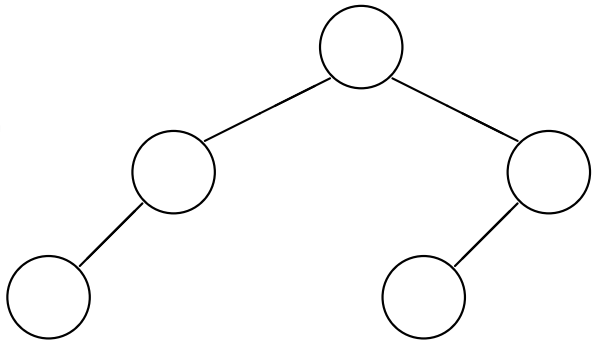
(2)



(3)



(4)



Welcher Baum hat keine Heap-Struktur?

Auflösung: (4)

Heapordnung für Max-Heaps

Ein geordneter binärer Baum T mit Heap-Struktur speichere für jeden Knoten v die Priorität $\mathbf{p(v)}$ von v. Dann hat T

Heap-Ordnung,

falls

$$\mathbf{p(v)} \geq \mathbf{p(w)}$$

für jeden Knoten v und für jedes Kind w von v gilt.

- Die höchste Priorität wird stets an der Wurzel gespeichert.
- Wie sollte man einen Baum mit Heap-Struktur implementieren? Wir arbeiten mit einem Array.

Der Heap

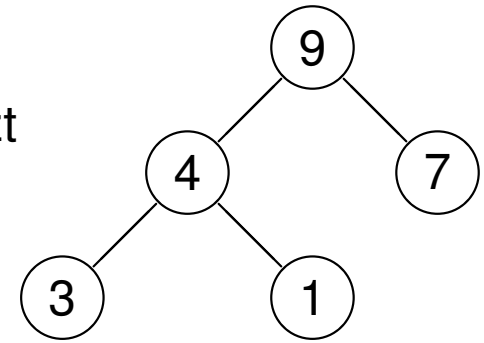
Die Datenstruktur Heap

Der geordnete binäre Baum T habe **Heap-Struktur** und **Heap-Ordnung**.

Das Array H ist ein **Heap** für T, wenn

- $H[1] = p(r)$ für die Wurzel r von T und
- wenn $H[i]$ die Priorität des Knotens v von T speichert, dann gilt
  - $\mathbf{H[2 \cdot i] = p(v_L)}$ für das linke Kind v_L von v und
  - $\mathbf{H[2 \cdot i + 1] = p(v_R)}$ für das rechte Kind v_R.

Zum Beispiel besitzt



den Heap $(9, 4, 7, 3, 1)$.

Die Funktion Insert

Wie **navigiert** man in einem Heap H?

- Wenn Knoten v in Position i gespeichert ist, dann ist
  - das linke Kind v_L in Position $2 \cdot i$,
  - das rechte Kind v_R in Position $2 \cdot i + 1$ und
  - der Elternknoten von v in Position $\lfloor i/2 \rfloor$

  gespeichert.

Wo sollten wir eine neue Priorität p einfügen?

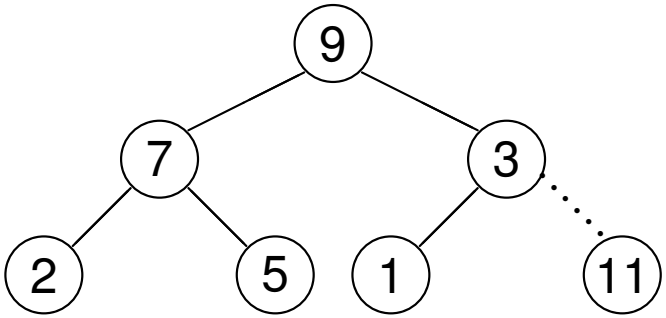
- Es liegt nahe, p auf der ersten freien Position abzulegen. Wir setzen also

$$H[++n] = p.$$

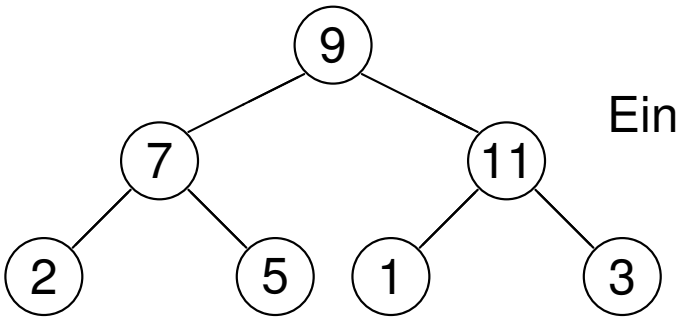
  - Der neue Baum hat Heap-Struktur, aber die Heap-Ordnung ist möglicherweise verletzt.

Wie kann die Heap-Ordnung kostengünstig repariert werden?

Wir fügen die Priorität 11 ein

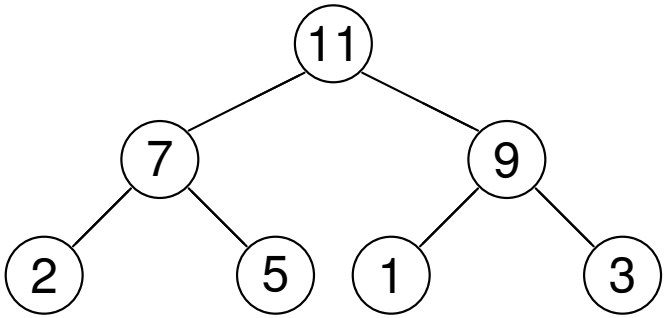


Die Heap-Ordnung ist verletzt und 11 rutscht nach oben:



Ein weiterer Vertauschungsschritt repariert die

Heap-Ordnung



Repair_up

Die Repair_up Prozedur

Die Klasse heap enthalte die Funktion **repair_up**.

```
void heap::repair_up (int wo)
  {int p = H[wo];
  while ((wo > 1) && (H[wo/2] < p))
       {H[wo] = H[wo/2];
       wo = wo/2; }
  H[wo] = p;}
```

- Wir verschieben die Priorität solange nach oben, bis
  - entweder die Priorität des Elternknotens mindestens so groß ist
  - oder wir die Wurzel erreicht haben.
- Wie groß ist der Aufwand?

Aufwand von Repair_up demogr.

Gegeben sei ein Heap mit n Elementen und Tiefe $t(n)$.

Dann benötigt Repair_up im Worst-Case die Zeit

- (1) $\Theta(1)$
- (2) $\Theta(\log t(n))$
- (3) $\Theta(\log \log n)$
- (4) $\Theta(t(n))$
- (5) $\Theta(\log^5 n + t(n))$
- (6) $\Theta(n + t(n))$

Auflösung: (4) $\Theta(t(n))$

Die Funktion Delete_max()

H repräsentiere einen Heap mit *n* Prioritäten. Für Delete_max:

1. Überschreibe die Wurzel mit $H[n]$
2. und verringere *n* um 1.

- Durch das Überschreiben mit $H(n)$ ist das entstandene Loch an der Wurzel verschwunden:

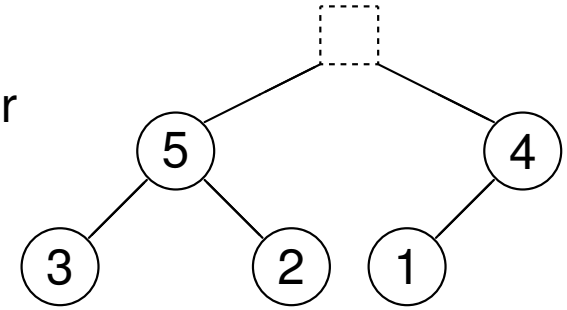
  Die Heap-Struktur ist wiederhergestellt.

- Allerdings ist die Heap-Ordnung möglicherweise verletzt und muss repariert werden.

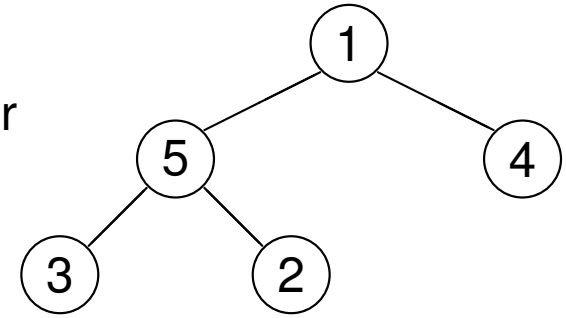
  Die Prozedur **repair_up** versagt: sie ist nur anwendbar, wenn die falsch stehende Priorität größer als die Eltern-Priorität ist.

Ein Beispiel

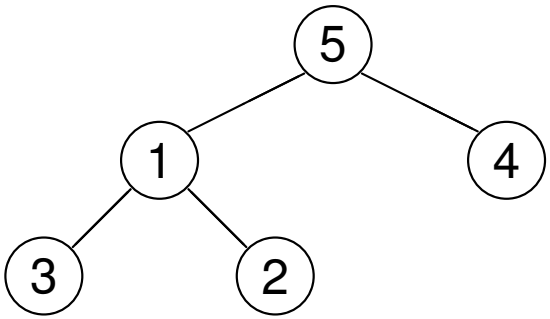
Vorher



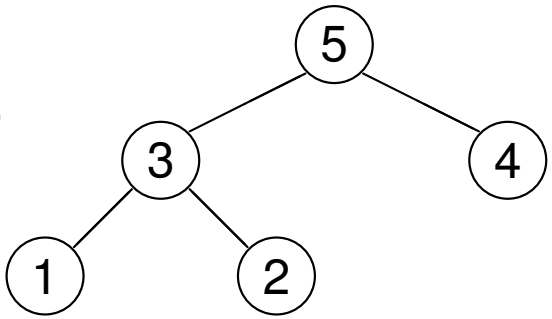
und nachher



Repariere die Heap-Ordnung: Vertausche mit dem **größtem** Kind



und wiederhole



und fertig.

Repariere die Heap-Ordnung nach unten.

Repair_down

Die Prozedur Repair_down

Die Klasse heap enthalte die Funktion **repair_down**.

```
void heap::repair_down (int wo)
  {int kind; int p = H[wo];
  while (wo <= n/2)
      {kind = 2 * wo;
      if ((kind < n) && (H[kind] < H[kind + 1])) kind ++;
      if (p >= H[kind]) break;
      H[wo] = H[kind]; wo = kind; }
  H[wo] = p; }
```

- Die Priorität *p* wird mit der Priorität des „größten Kinds“ verglichen und möglicherweise vertauscht.
- Die Prozedur endet, wenn wo die richtige Position ist, bzw. wenn wo ein Blatt beschreibt.
- Wie groß ist der Aufwand? Höchstens proportional zur Tiefe.

Change_priority und Remove

- **void change_priority** (int *wo*, int *p*):
  - Wir aktualisieren die Priorität, setzen also H[*wo*] $= p$.
  - Aber wir verletzen damit möglicherweise die Heap-Ordnung!
    - ⋆ Wenn die Priorität angewachsen ist, dann rufe **repair_up** auf.
    - ⋆ Ansonsten hat sich die Priorität verringert und **repair_down** ist aufzurufen.
- **void remove**(int *wo*):
  - Stelle die Heap-Struktur durch H[*wo*] $=$ H[*n*- -]; wieder her
  - und rufe dann change_priority auf.

Alle vier Operationen

insert, delete_max, change_priority und **remove**

benötigen Zeit höchstens proportional zur Tiefe des Heaps.

Die Tiefe eines Heaps mit n Knoten

Der Binärbaum T besitze Heap-Struktur.

- Wenn T die Tiefe $t = \mathit{Tiefe}(T)$ besitzt, dann hat T mindestens

$$1 + 2^1 + 2^2 + \ldots + 2^{t-1} + 1 = 2^t$$

  Knoten
- aber nicht mehr als

$$1 + 2^1 + 2^2 + \ldots + 2^{t-1} + 2^t = 2^{t+1} - 1$$

  Knoten.
- Also folgt für die Knotenzahl n,

$$2^{\mathit{Tiefe}(T)} \leq n < 2^{\mathit{Tiefe}(T)+1}.$$

Es ist

$$\mathit{Tiefe}(T) = \lfloor \log_2 n \rfloor$$

und alle vier Operationen werden somit in logarithmischer Zeit unterstützt!

Heapsort

Ein Array $(A[1], \ldots, A[n])$ ist zu sortieren.

```
for (i=1; i <= n ; i++)
    insert(A[i]);
// buildheap ist schneller
int N = n;
for (n=N; n >= 1 ; n- -)
    A[n] = delete_max( );
//Das Array A ist jetzt aufsteigend sortiert.
```

- Zuerst werden n Schlüssel eingefügt und dann wird n Mal das Maximum entfernt.
- Sowohl die anfängliche Einfügephase wie auch die letztliche Entfernungsphase benötigen Zeit höchstens $O(n \cdot \log_2 n)$.

Heapsort ist eines der schnellsten Sortierverfahren.

Wie kann der Heap schneller geladen werden?

Führe statt vielen kleinen Reparaturen eine große Reparatur durch.

- Beginne die Reparatur mit den Blättern. Jedes Blatt ist schon ein Heap und eine Reparatur ist nicht notwendig.
- Wenn t die Tiefe des Heaps ist, dann kümmern wir uns als Nächstes um die Knoten v der Tiefe $t - 1$.
  - Sei T_v der Teilbaum mit Wurzel v.
  - T_v ist nur dann kein Heap, wenn die Heap-Ordnung im Knoten v verletzt ist: Repariere mit repair_down, gestartet in v.
  - Höchstens ein Vertauschungsschritt wird benötigt.
- Wenn v ein Knoten der Tiefe $t - j$ ist, dann muss höchstens die Heap-Ordnung im Knoten v repariert werden.
  - Höchstens j Vertauschungsschritte genügen.

Es gibt nur wenige teure Reparaturschritte!

Analyse

- Es gibt 2^{t-j} Knoten der Tiefe $t-j$ (für $j \geq 1$).
- Für jeden dieser Knoten sind höchstens j Vertauschungsschritte durchzuführen und die Ladezeit ist durch $\sum_{j=1}^{t} j \cdot 2^{t-j}$ beschränkt.
- Behauptung: $\sum_{j=1}^{t} j \cdot 2^{t-j} = 2^{t+1} - t - 2$. Wir geben einen induktiven Beweis:

$$\begin{aligned}
\sum_{j=1}^{t+1} j \cdot 2^{t+1-j} &= 2 \sum_{j=1}^{t} j \cdot 2^{t-j} + t + 1 \\
&= 2 \cdot (2^{t+1} - t - 2) + t + 1 \\
&= 2^{t+2} - (t+1) - 2.
\end{aligned}$$

Der Heap kann in linearer Zeit geladen werden.

Die Klasse heap

```
class heap
{private:
    int *H; // H ist der Heap.
    int n; // n bezeichnet die Größe des Heaps.
    void repair_up (int wo);
    void repair_down (int wo);
public:
    heap (int max) // Konstruktor.
        { H = new int[max]; n = 0; }
    int read (int i) { return H[i]; }
    void insert (int priority);
    int delete_max( );
    void change_priority (int wo, int p);
    void remove(int wo);
    void buildheap();
    void heapsort(); };
```

Prioritätswarteschlangen: Zusammenfassung

(a) Ein Heap mit n Prioritäten unterstützt jede der Operationen

insert, **delete_max**, **change_priority** und **remove**

in Zeit $O(\log_2 n)$.

- Für die Operationen **change_priority** und **remove** muss die Position der zu ändernden Priorität bestimmt werden. (Übungsaufgabe)

(b) **buildheap** baut einen Heap mit n Prioritäten in Zeit $O(n)$.

(c) **heapsort** sortiert n Zahlen in Zeit $O(n \log_2 n)$.