ILV Datenstrukturen und Algorithmen

07: Stacks und Queues

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

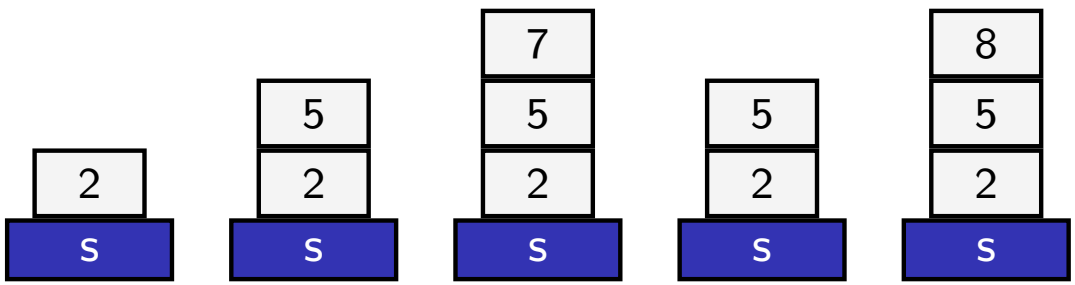
Kapitel

Stacks und Queues

Stacks: Definition

- Ein *stack* (a.k.a. pushdown stack, Kellerspeicher) ist eine einfache Datenstruktur, die nur zwei (drei) grundlegende Operationen definiert:
  - `push`: Ein Datenelement ablegen
  - `pop`: Das zuletzt eingefügte Datenelement wieder entfernen
  - (`top`: Liefert das zuletzt eingefügte Datenelement)
- Ein Stack ist ein sogenannter *LIFO*-Speicher:
  - Last-in first-out
  - Das Letzte das 'rein kam kommt als Ersteres wieder raus.

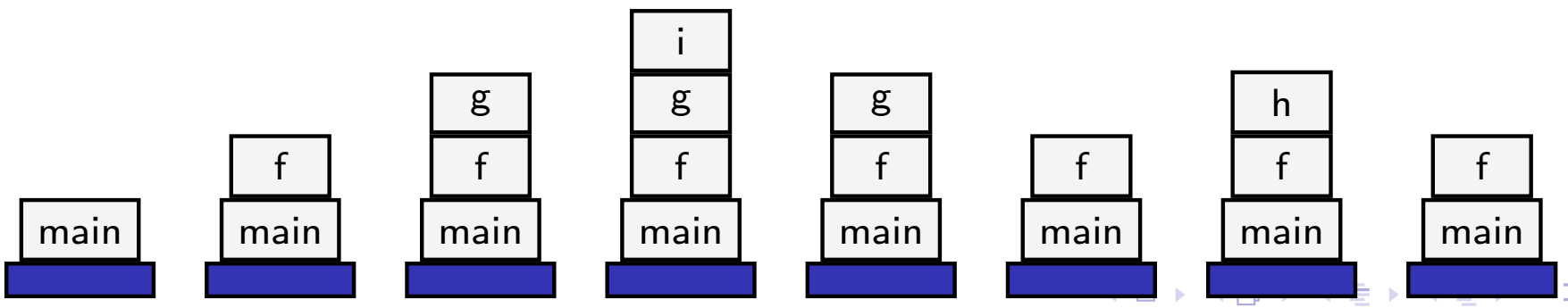
```
1 Stack s;
2 s.push(2);
3 s.push(5);
4 s.push(7);
5 s.pop();
6 s.push(8);
```



Anwendungen: function calls und call stack

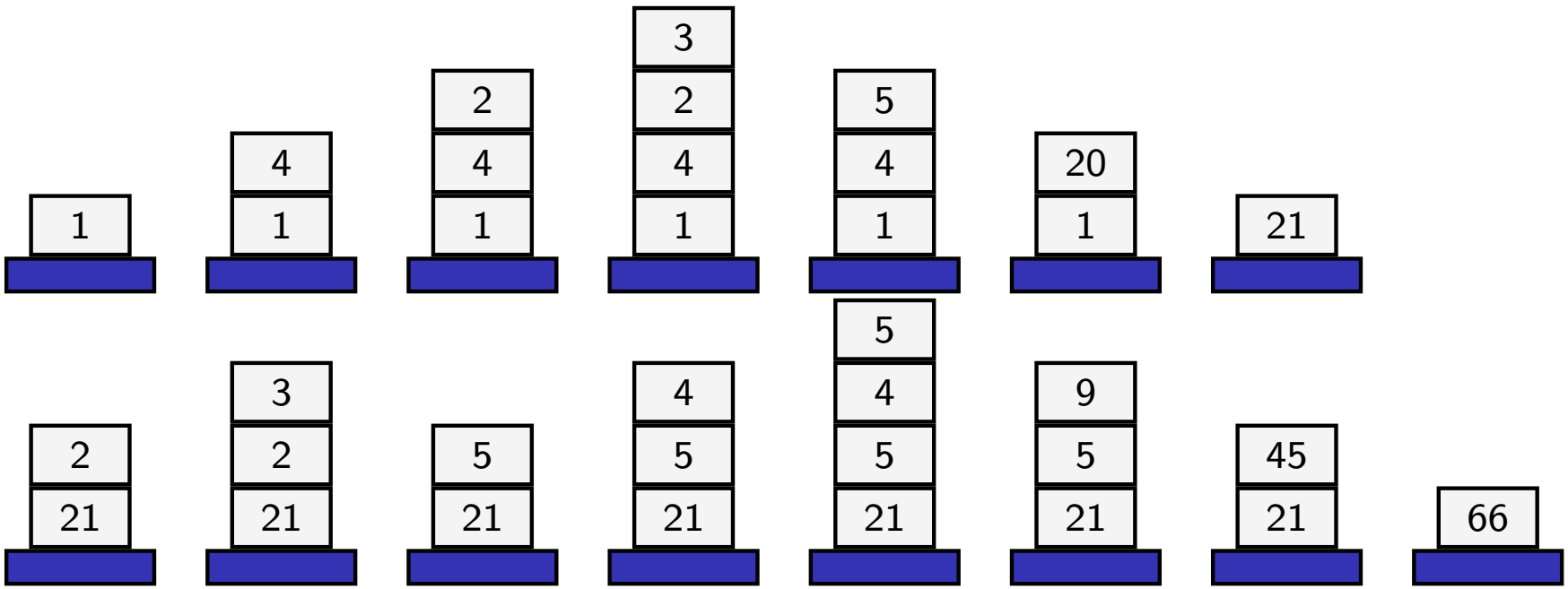
- Lokale Variablen und Argumente einer C-Funktionen liegen auf einem Stack, dem sogenannten *call stack*.
  - `f()`: entspricht einem `push()` auf diesem Stack.
  - `return`: entspricht einem `pop()` auf diesem Stack.

```
1  void f()
2  { g(); h(); }
3  void g()
4  { i(); }
5  void h()
6  { }
7  void i()
8  { }
9  int main()
10 {   f(); return 0;  }
```



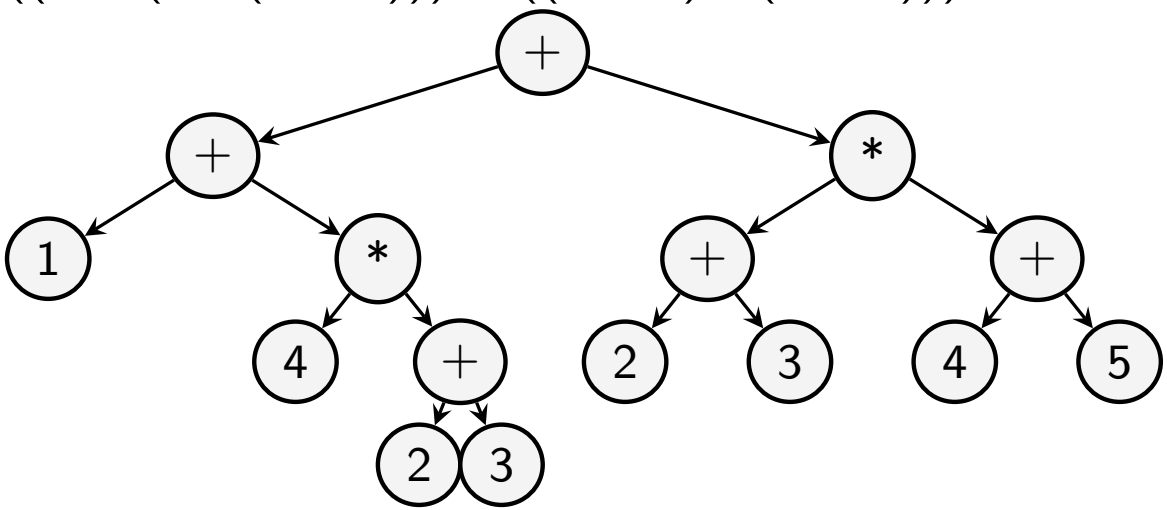
Anwendungen: Arithmetische Ausdrücke

- Stacks werden häufig dazu verwendet um Zwischenergebnissen abzuspeichern.
- Beispiel: $((1 + (4 * (2 + 3))) + ((2 + 3) * (4 + 5))) = 66$
  - Wenn wir eine Zahl sehen, geben wir diese auf den Stack.
  - Wenn eine Klammer zugeht, dann nehmen wir die beiden letzten Zahlen vom Stack, führen die zugehörige Operation in der Klammer aus, und geben das Ergebnis wieder auf den Stack.



Anwendungen: Arithmetische Ausdrücke

- Wie sehe ich, welche Operation in einer Klammer steht?
  - Die postfix-Notation (a.k.a. umgekehrte polnische Notation) für arithmetische Ausdrücke hilft hier.
  - Man schreibt den Operator *nach* den Operanden: $((A) + (B))$ wird zu $AB+$.
  - $((2 + 3) * (4 + 5))$ wird zu $23 + 45 + *$.
    - $(2 + 3)$ wird zu $23+$ und $(4 + 5)$ wird zu $45+$.
- Allgemein: man betrachtet den sogenannten *Syntaxbaum* zum infix Ausdruck $((1 + (4 * (2 + 3))) + ((2 + 3) * (4 + 5)))$



- Postfix Notation: 1423+*+23+45+*+

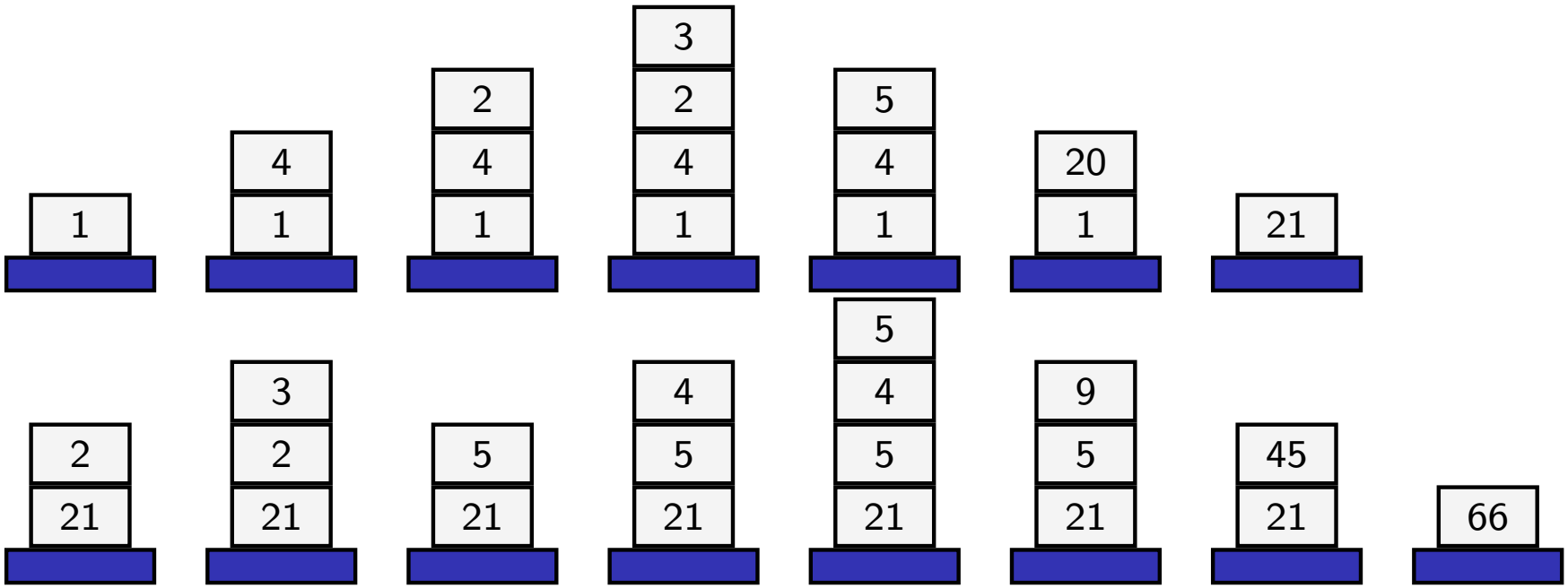
Anwendungen: Arithmetische Ausdrücke

- Ein postfix Ausdruck lässt sich leicht berechnen:
  - Wir laufen von links nach rechts. Wenn wir einen Operator sehen, dann (i) nehmen wir die beiden vorherigen Zahlen, (ii) wenden den Operator an und (iii) ersetzen die beiden Zahlen und den Operator durch das Ergebnis.

    1 4 **2 3 +** * + 2 3 + 4 5 + * +
    1 **4 5 *** + 2 3 + 4 5 + * +
    1 20 + 2 3 + 4 5 + * +
    21 **2 3 +** 4 5 + * +
    21 5 **4 5 +** * +
    21 **5 9 *** +
    21 45 +
    66
- Ist dieses Verfahren **korrekt**?
  - **Schleifeninvariante:** Ein Schritt verändert das Ergebnis des Ausdrucks nicht! (Warum?) Das Ergebnis des Ausdrucks ist *invariant*.
- **Terminiert** dieses Verfahren?
  - In jedem Schritt wird der Ausdruck um eine Zahl und einen Operanden kürzer. Am Ende bleibt nur noch eine Zahl.
- Wie wandelt man dieses Verfahren in einen Algorithmus um?
  - Verwende einen Stack für alle Zahlen bis zum ersten Operator.

Anwendungen: Arithmetische Ausdrücke

- Algorithmus:
  - Wenn wir eine Zahl sehen, geben wir diese auf den Stack.
  - Wenn wir einen Operator sehen, dann nehmen wir die beiden letzten Zahlen vom Stack, wenden den Operator an, und geben das Ergebnis auf den Stack.
- 1 4 2 3 + * + 2 3 + 4 5 + * +

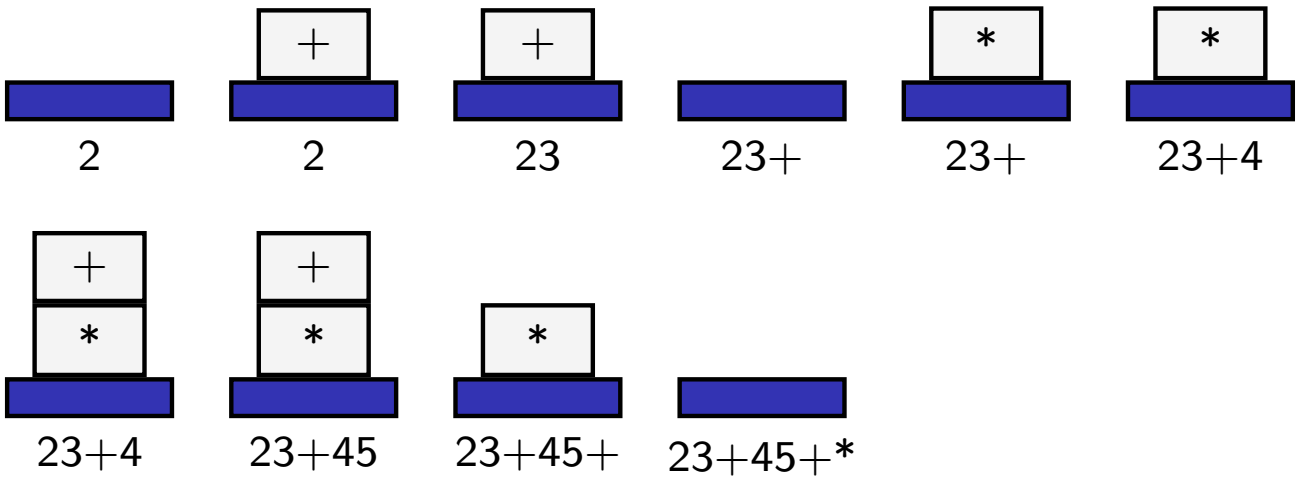


Anwendungen: Postfix Ausdruck berechnen

```
1  int computePostfix(const string& postfix)
2  {
3    Stack<int> s;
4    for(unsigned i=0; i<postfix.size(); ++i)
5    {
6      const char ch = postfix[i];
7      if( ch=='+' || ch=='*' )
8      {
9        const int a = s.top(); s.pop();
10       const int b = s.top(); s.pop();
11       if( ch=='+' )
12         s.push(a+b);
13       else if( ch=='*' )
14         s.push(a*b);
15     }
16     else if( '0' <= ch && ch <= '9' )
17       s.push(ch - '0');
18   }
19   return s.top();
20 }
```

Anwendungen: Infix zu Postfix

- Problemstellung
  - Input: ein vollständig geklammerter[1], syntaktisch korrekter infix-Ausdruck, etwa $((2 + 3) * (4 + 5))$.
  - Output: der entsprechende postfix-Ausdruck
- Idee:
  - Gib Operanden (Zahlen) sofort aus.
  - Merke dir die Operatoren auf dem Stack und gib einen aus, wenn ein) kommt.
  - Ein (wird ignoriert.



[1] Zu jedem Operator gehört genau ein Paar von Klammern.

Anwendungen: Infix zu Postfix

```
1  string infix2postfix(const string& infix)
2  {
3    stringstream ss;
4    Stack<char> s;
5    for(unsigned i=0; i<infix.size(); ++i)
6    {
7      const char ch = infix[i];
8      if( ch=='+' || ch=='*' )
9        s.push(ch);
10     else if( '0' <= ch && ch <= '9' )
11       ss << ch;
12     else if( ch==')' )
13     {
14       ss << s.top();
15       s.pop();
16     }
17   }
18   return ss.str();
19 }
```

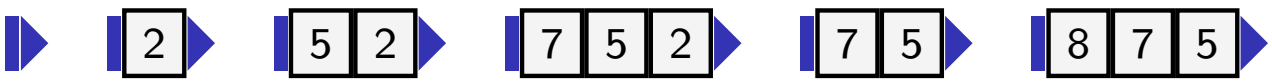
Implementierung des Stacks

- Einfach verkettete Liste `SList l`:
  - `pop()`: `l.pop_front()`
  - `push(value)`: `l.push_front(value)`
  - `top()`: `l.front()`
- Wenn die maximale Größe des Stacks bekannt ist, dann kann ein Array zielführend sein.
  - Man unterhält einen Index `idx`, der die Position des top-Elements des Stacks speichert.
  - `pop()`: `idx--;`
  - `push(value)`: `idx++; stack[idx]=value;`
  - `top()`: `return stack[idx];`

Queues: Definition

- Neben Stacks sind *Queues* (a.k.a. Warteschlange) eine weitere grundlegende Datenstruktur mit nur zwei (drei) Operationen:
  - enqueue/insert: Ein Datenelement ablegen
  - dequeue/remove: Das älteste Datenelement entfernen
  - (front/get: Liefert das älteste Datenelement)
- Eine Queue ist ein sogneannter FIFO-Speicher:
  - First-in first-out
  - Das Erste das 'rein kommt, kommt als Ersteres wieder raus.
  - A.k.a. FCFS — first come first serve

```
1 Queue q;
2 q.enqueue(2);
3 q.enqueue(5);
4 q.enqueue(7);
5 q.dequeue();
6 q.enqueue(8);
```



Anwendungen

- Kommunikation zwischen asynchronen Prozessen:
  - Ein Prozess A erzeugt Daten.
  - Ein Prozess B liest Daten von A.
  - Die beiden Prozesse erzeugen/lesen die Daten nicht synchron.
  - Idee: A schreibt in eine Queue und B liest davon.
  - (Zusätzliches Problem: concurrent programming)
- Allgemeiner: n parallele Prozesse $A_1, \ldots, A_n$ erzeugen Daten, welche von m parallelen Prozessen $B_1, \ldots, B_m$ verarbeitet werden.
  - Man hat wieder eine Queue, in welche $A_1, \ldots, A_n$ Daten hineingeben und $B_1, \ldots, B_m$ Daten davon lesen.
- Ähnlich: Message-Queues in graphischen Anwendungen.
- Später: Level-order Traversierung von Binärbäumen oder breadth-depth search in Graphen.

Implementierung einer Queue

- Zweifach verkettete Liste `List l`:
  - `enqueue(value)`: `l.push_back(l)`
  - `dequeue()`: `l.pop_front()`
  - `front()`: `l.front()`
- Einfach verkettete Liste mit zusätzlichem tail pointer. (Erlaubt das einfache Einfügen am Ende.)
- Für Interprozesskommunikation ist die Größe der Queue oft beschränkt (Puffergröße gegeben).
  - Queue in ein Array ablegen.
  - Als sogenannten *circular buffer*.
  - Man hat zwei Indizes (head und tail), welche im Array „im Kreis“ laufen.

Circular buffer

```
1  template<class T>  class CircularBuffer
2  {
3      T* buf; unsigned head, tail, capac;
4    public:
5      CircBuffer(unsigned size): head(0), tail(0), capac(size+1)
6      {
7        buf = new T[capac];
8      }
9      void enqueue(const T& val)
10     {
11       // tail==head means empty or full? -> leave one slot empty
12       assert( count() < capac-1 );
13       buf[head] = val;
14       head = (head+1)%capac;
15     }
16     T dequeue()
17     {
18       assert( count() > 0 );
19       T t = buf[tail];
20       tail = (tail+1)%capac;
21       return t;
22     }
23     unsigned count() const { return (head-tail+capac)%capac; }
24 };
```

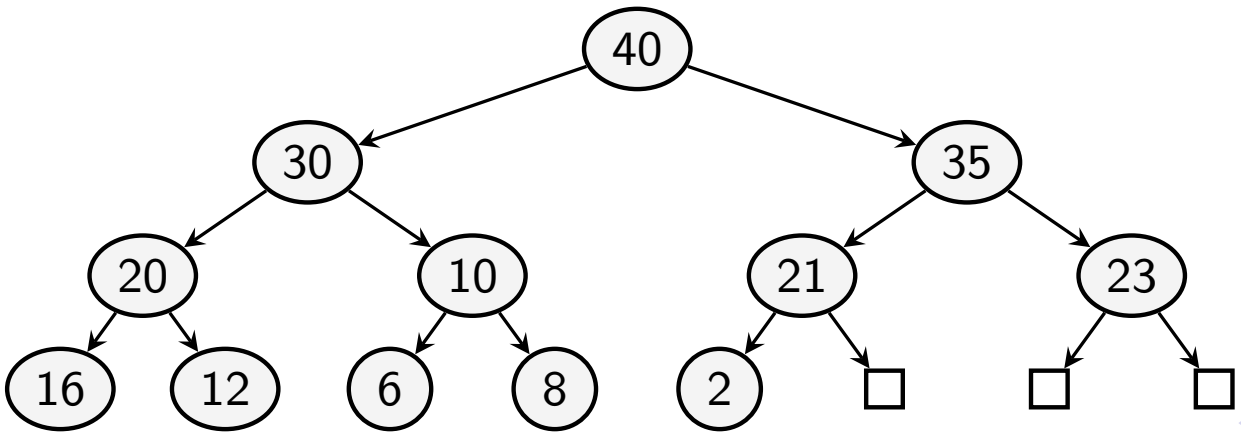
Variationen der gewöhnlichen Queue

- *Deque*: double-ended queue
  - Erlaubt Einfügen und Entfernen an beiden Enden.
  - Über eine doppelt verkettete List einfach zu haben.
- *Priority queue* (a.k.a. Prioritätswarteschlangen)
  - Jedes Element hat eine Priorität
  - `dequeue()` entfernt nicht das älteste Element, sondern jenes mit der höchsten (kleinsten) Priorität.
  - Typisches Beispiel: Event-Simulation, Priorität entspricht dem Zeitpunkt des Events, welche nicht chronologisch eingefügt werden müssen, aber chronologisch entfernt werden.
  - Anforderung: `enqueue`, `dequeue` und `front` sollen sehr schnell gehen.
  - Möglich: Angenommen eine Warteschlange hat 10^6 Einträge. `enqueue` und `dequeue` in ca. 20 Schritten[2] und `front` in 1 Schritt.

[2] Ein Schritt heißt, soviele Einträge in der Queue muss ich betrachten/vertauschen.

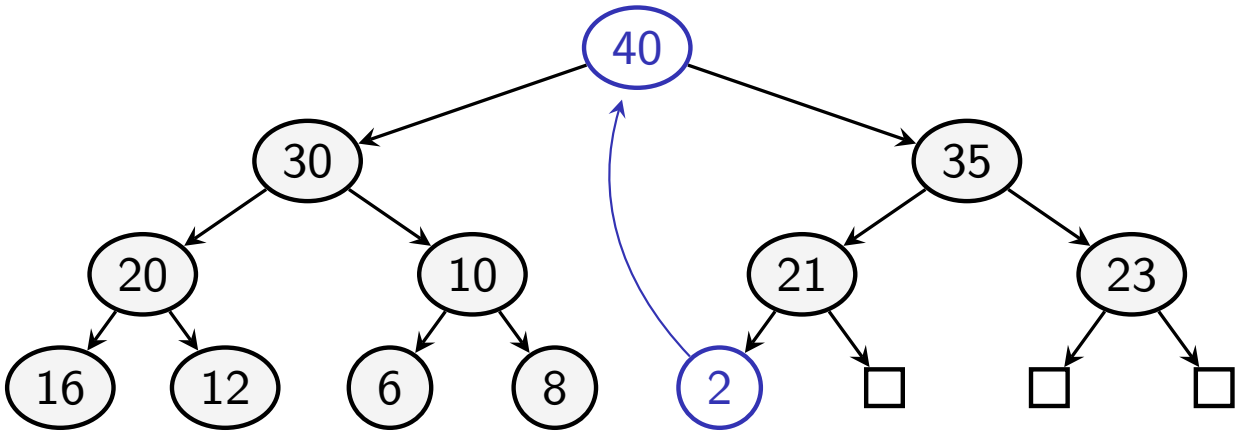
Heaps

- Prioritätswarteschlangen realisiert man effizient über sogenannte Heaps.
- Die Daten werden in einem Binärbaum organisiert.
  - Jeder Knoten hat zwei Pointer: auf einen linken und einen rechten Kinderknoten.
  - Der Knoten ganz oben heißt *Wurzel*.
    - Bis auf die Wurzel hat jeder Knoten einen Elternknoten.
  - Ein *Level* besteht aus allen Knoten auf „gleicher Höhe“.
- Ein Binärbaum heißt Max-Heap (maximizing heap),
  1. wenn für jeden Knoten gilt, dass dessen Kinder kleinere Werte (oder den gleichen) speichern und
  2. wenn alle Level voll sind, bis auf das unterste Level, welches von links her gefüllt ist.



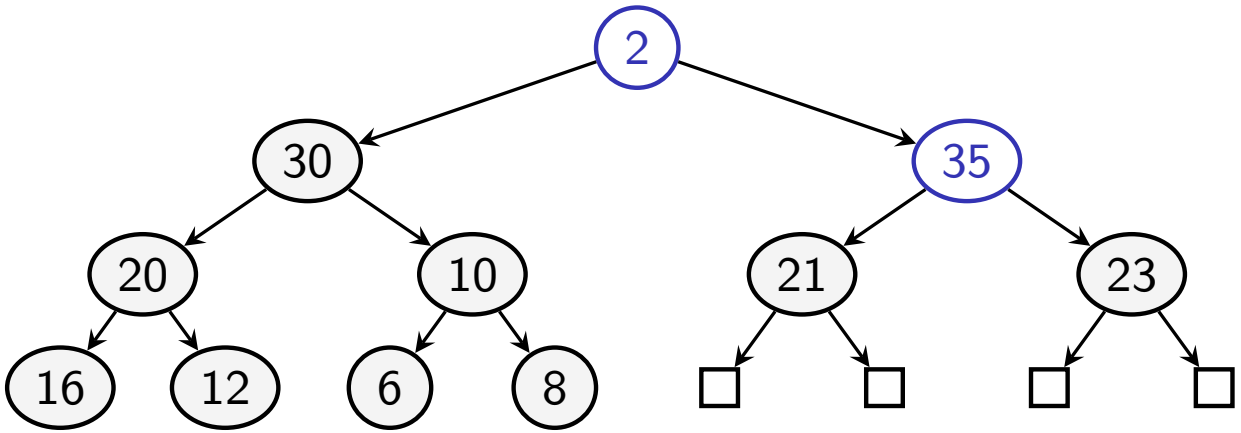
Heap: dequeue

- Wir entfernen das größte Element, welches per Definition in der Wurzel liegt.
  1. Nimm vom untersten Level das rechteste Element und gib es an die Wurzel.
  2. Stelle die Heap-Eigenschaft wieder her: Tausche das größere Kind der Wurzel mit der Wurzel und wiederhole dies für das getauschte Kind, solange es ein größeres Kind gibt.



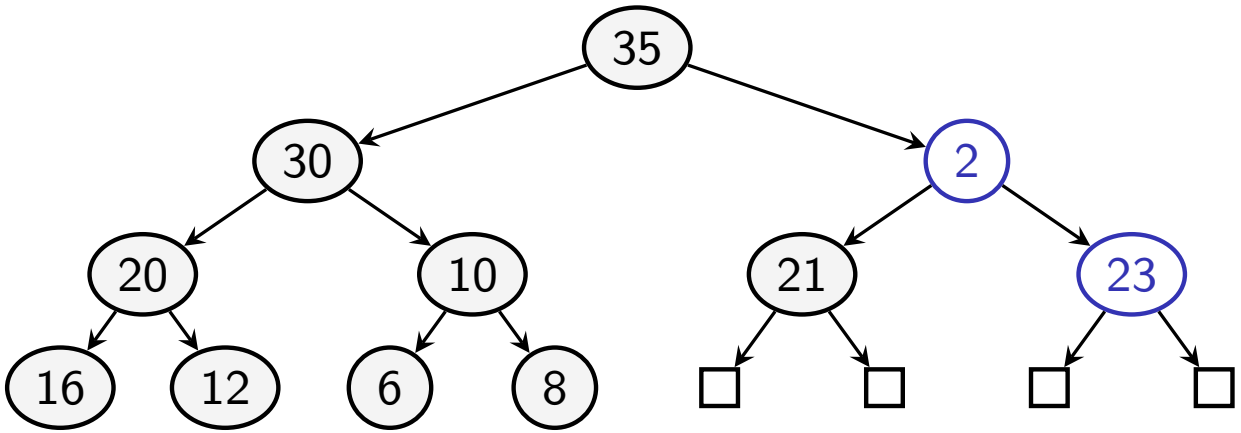
Heap: dequeue

- Wir entfernen das größte Element, welches per Definition in der Wurzel liegt.
  1. Nimm vom untersten Level das rechteste Element und gib es an die Wurzel.
  2. Stelle die Heap-Eigenschaft wieder her: Tausche das größere Kind der Wurzel mit der Wurzel und wiederhole dies für das getauschte Kind, solange es ein größeres Kind gibt.



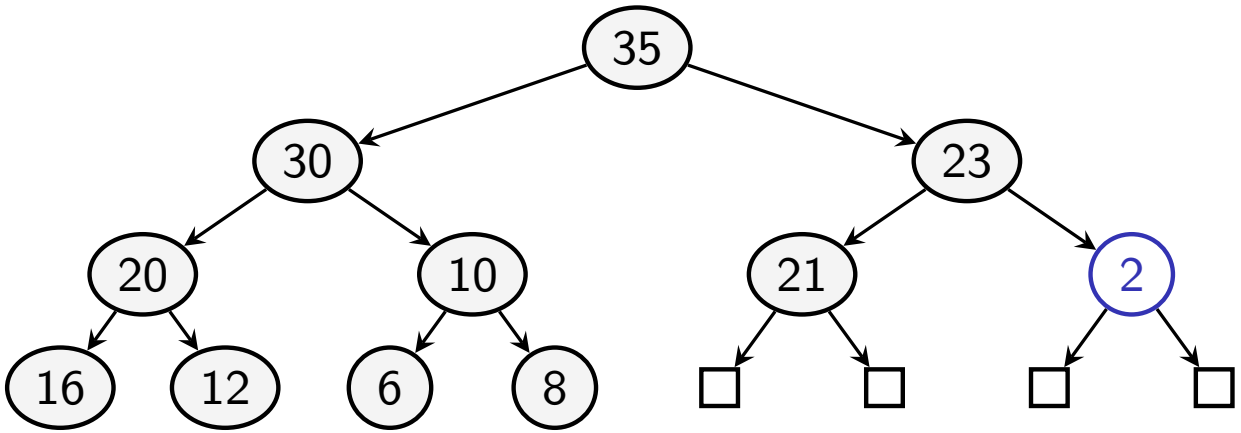
Heap: dequeue

- Wir entfernen das größte Element, welches per Definition in der Wurzel liegt.
  1. Nimm vom untersten Level das rechteste Element und gib es an die Wurzel.
  2. Stelle die Heap-Eigenschaft wieder her: Tausche das größere Kind der Wurzel mit der Wurzel und wiederhole dies für das getauschte Kind, solange es ein größeres Kind gibt.



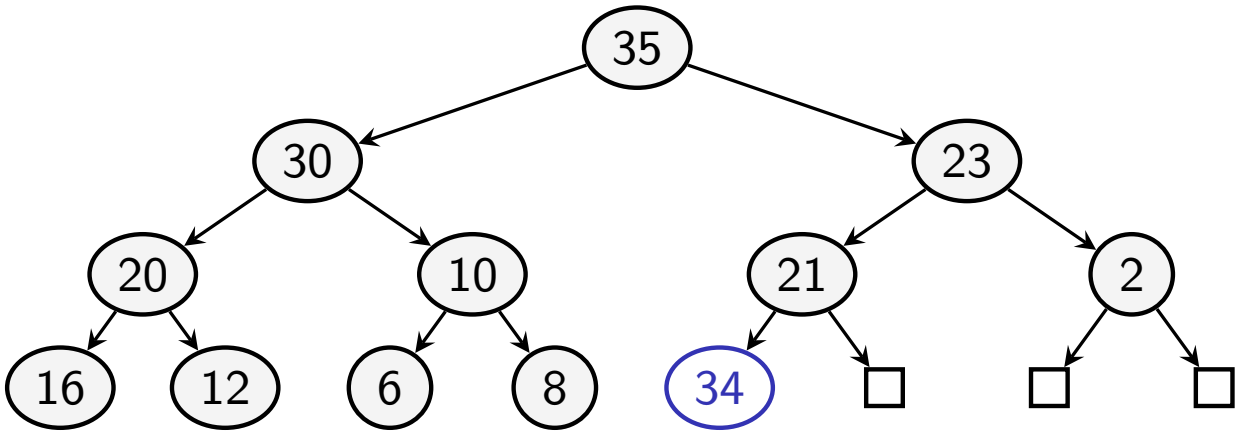
Heap: dequeue

- Wir entfernen das größte Element, welches per Definition in der Wurzel liegt.
  1. Nimm vom untersten Level das rechteste Element und gib es an die Wurzel.
  2. Stelle die Heap-Eigenschaft wieder her: Tausche das größere Kind der Wurzel mit der Wurzel und wiederhole dies für das getauschte Kind, solange es ein größeres Kind gibt.



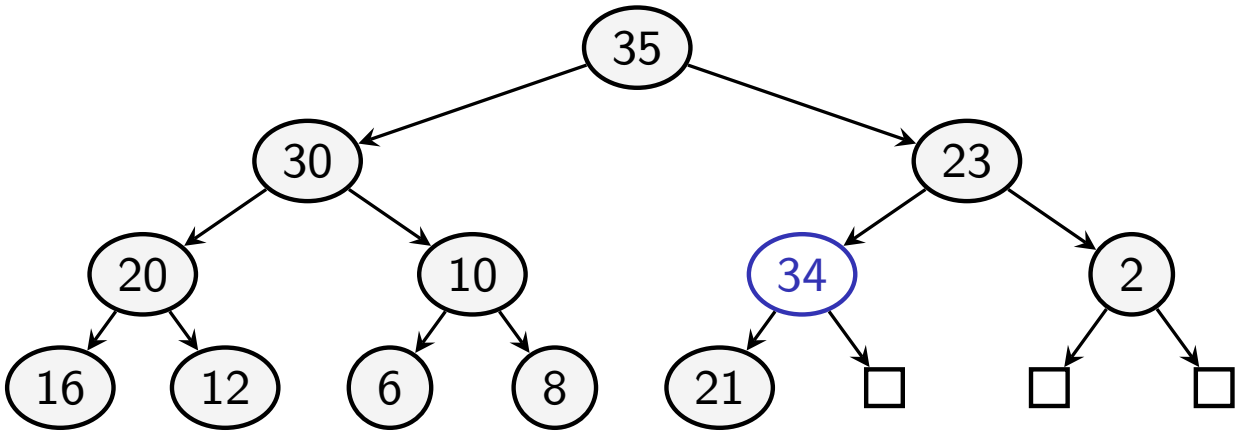
Heap: enqueue

- Wir fügen ein neues Element in den Heap ein.
  1. Gib das Element in das unterste Level ganz rechts (bzw. starte ein neues Level, wenn das unterste Level voll ist.)
  2. Stelle die Heap-Eigenschaft wieder her: Tausche den eingefügten Knoten mit seinem Elternknoten, solange dieser größer ist.



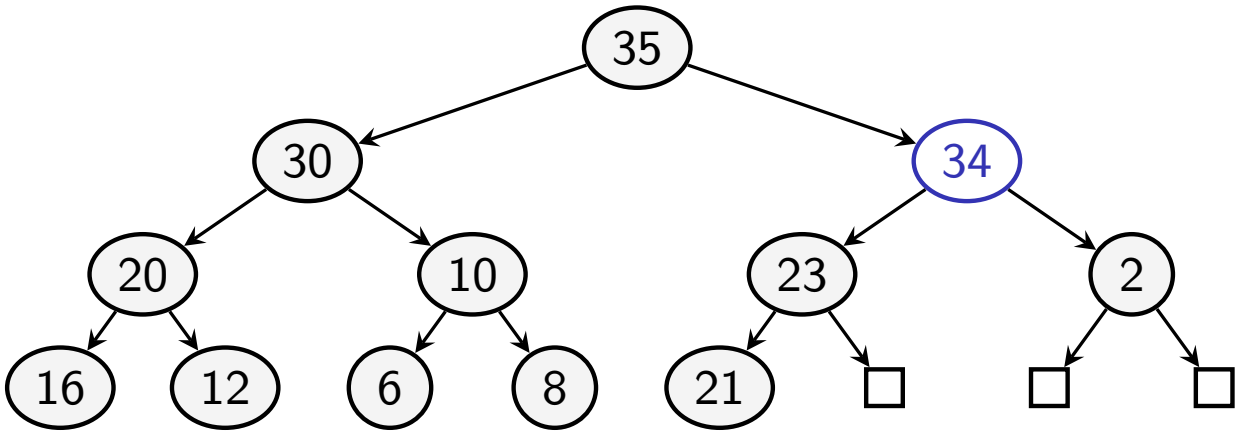
Heap: enqueue

- Wir fügen ein neues Element in den Heap ein.
  1. Gib das Element in das unterste Level ganz rechts (bzw. starte ein neues Level, wenn das unterste Level voll ist.)
  2. Stelle die Heap-Eigenschaft wieder her: Tausche den eingefügten Knoten mit seinem Elternknoten, solange dieser größer ist.



Heap: enqueue

- Wir fügen ein neues Element in den Heap ein.
  1. Gib das Element in das unterste Level ganz rechts (bzw. starte ein neues Level, wenn das unterste Level voll ist.)
  2. Stelle die Heap-Eigenschaft wieder her: Tausche den eingefügten Knoten mit seinem Elternknoten, solange dieser größer ist.

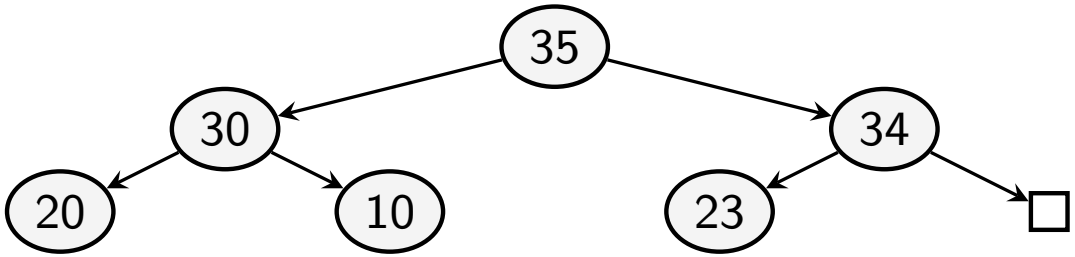


Heap: Effizienz

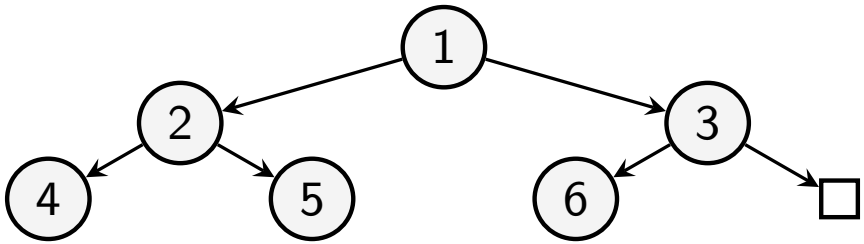
- Wie schnell geht `enqueue` und `dequeue`?
  - Wir brauchen höchstens soviele Vertauschungen wie der Heap Levels hat.
  - Der Heap hat höchstens $1 + \log_2 n$ Levels, wenn n die Zahl der Knoten bezeichnet.
- Wie schnell geht `front`?
  - Einach die Wurzel zurückliefern.

Heap: Implementierung

- Heaps werden effizient in Arrays abgespeichert, indem man Level für Level von links her in das Array gibt.
- Das Array [35 30 34 20 10 23] entspricht dem folgendem Heap:



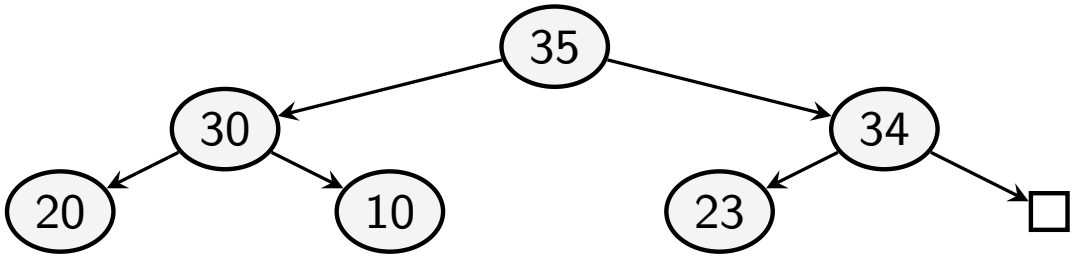
- Die Indizes der Knoten im Array entsprechen also den folgenden Werten.
  - Aus Bequemlichkeit beginnen wir bei Index 1.



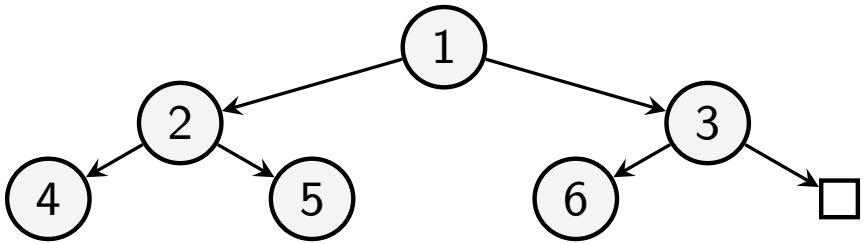
- Betrachten wir den Knoten mit Index i:
  - Der Elternknoten ist bei Index $\lfloor i/2 \rfloor$.
  - Die Kinder sind bei den Indizes $2i$ und $2i + 1$.

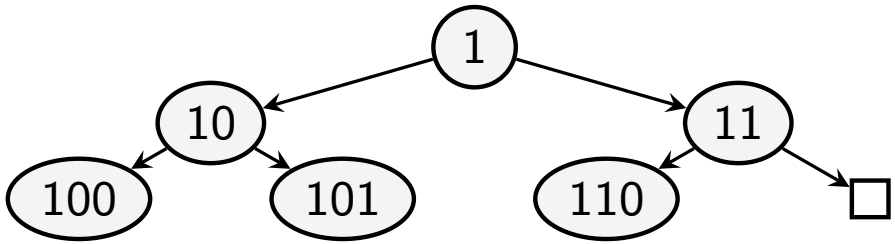
Heap: Implementierung

- Heaps werden effizient in Arrays abgespeichert, indem man Level für Level von links her in das Array gibt.
- Das Array [35 30 34 20 10 23] entspricht dem folgendem Heap:



- Die Indizes der Knoten im Array entsprechen also den folgenden Werten.
  - Aus Bequemlichkeit beginnen wir bei Index 1.





- Betrachten wir den Knoten mit Index i:
  - Der Elternknoten ist bei Index $\lfloor i/2 \rfloor$.
  - Die Kinder sind bei den Indizes $2i$ und $2i + 1$.

Abstrakte Datentypen

- Man betrachte nicht die konkrete Implementierung eine Datenstruktur, sondern nur die Operationen (`push`, `dequeue`, ...) mit deren Semantik und und deren Zeit-/Speicher-Charakteristik.
- Ein Stack ist „etwas“, dass `push` und `pop` in konstanter Zeit kann und dessen Speicherverbrauch im gleichen Ausmaß wie die Zahl der gespeicherten Elemente steigt.
  - Im Hintergrund kann man das mit einer einfach verketteten Liste realisieren.
  - Für die Auswertung von arithmetischen Ausdrücken interessiert das aber nicht.
- Günstig für die Analyse und Entwicklung von Algorithmen.
  - Man konzentriert sich auf das Verhalten von Datenstrukturen, nicht auf deren Implementierung.