Sortieralgorithmen

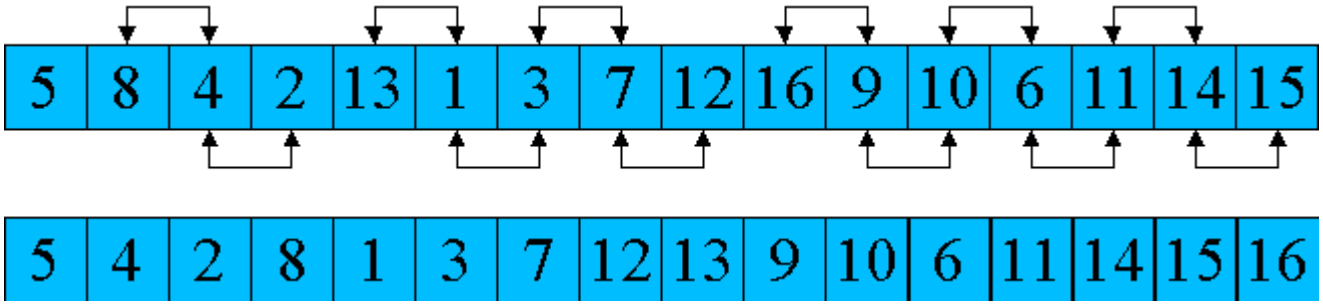
Direkte Sortierverfahren
&
Shellsort, Quicksort, Heapsort

Vorlesung
Algorithmen und Datenstrukturen 2
im SS 2004

Prof. Dr. W. P. Kowalk
Universität Oldenburg

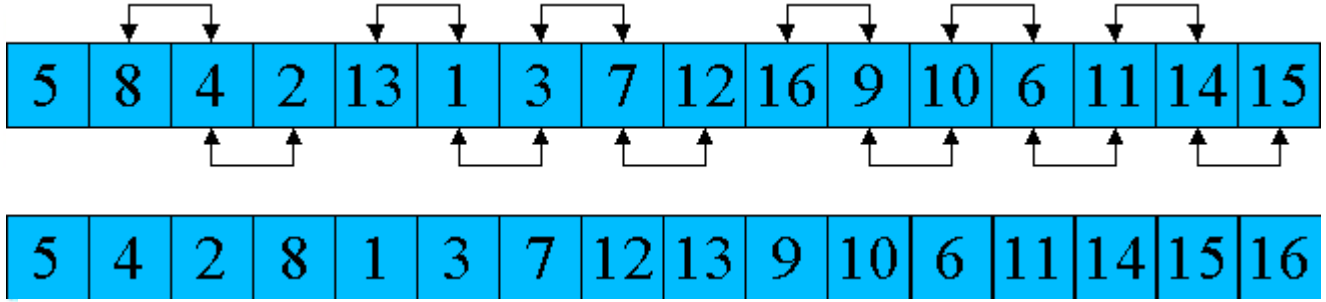
Sortieren durch Vertauschen

- Einfachstes direkte Sortierverfahren
- vertauscht zwei benachbarte Elemente, die in falscher Reihenfolge stehen
- bis alle Elemente in richtiger Reihenfolge stehen
- Aufwand = Fehlstellungszahl = $O(N^2)$



Direktes Vertauschen – 1

- Sortieren durch Vertauschen benachbarter Elemente



- Aufwand
  - Vergleiche: $V = (N-1)^2$
  - Vertauschungen: Anzahl der Fehlstellungen

Direktes Vertauschen – 2

- Implementierung von **BubbleSort**

```
static public void BubbleSort(CDatenSatz[] Feld) {
  for (int zähler = 1; zähler < Feld.length; zähler++) {
    // length-1 Mal
    for (int index = 1; index < Feld.length; index++) {
      // Indizes 1, 2, .. lenght-1
      if (Feld[index - 1].key > Feld[index].key) {
        // vertausche
        CDatenSatz swap = Feld[index - 1];
        Feld[index - 1] = Feld[index];
        Feld[index] = swap;
} } } }
```

Direktes Vertauschen – 3

- Implementierung von **Vertauschen**
  - Verringern der oberen Feldgrenze

```
static public void Vertauschen(CDatenSatz[] Feld) {
  for (int grenze = Feld.length - 1;
       grenze >= 1; grenze--) {
    // length-1 Mal
    for (int index = 1; index <= grenze; index++) {
      // Indizes 1, 2, .. grenze
      if (Feld[index-1].key > Feld[index].key) {
        // vertausche
        CDatenSatz swap = Feld[index];
        Feld[index] = Feld[index-1];
        Feld[index-1] = swap;
  } } } }
```

Direktes Vertauschen – 4

- Implementierung von **Vertauschen**
  - Verringern der oberen Feldgrenze
  - Beende, falls keine weitere Vertauschung

```
static public void Vertauschen(CDatenSatz[] Feld) {
  for (int grenze = Feld.length - 1;
           grenze >= 1; grenze--) {
    // length-1 Mal
    boolean wurdeVertauscht = false;
    for (int index = 1; index <= grenze; index++) {
    // Indizes 1, 2, .. grenze
      if (Feld[index-1].key > Feld[index].key) {
        // vertausche
        CDatenSatz swap = Feld[index];
        Feld[index] = Feld[index-1];
        Feld[index-1] = swap;
        wurdeVertauscht = true;
    } }
    if(!wurdeVertauscht) break;
} }
```

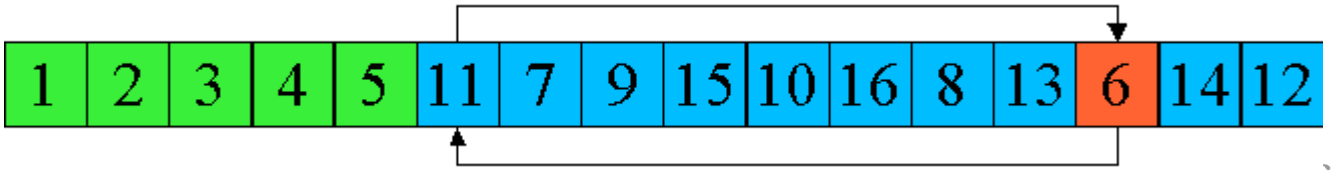
Direktes Auswählen – 1

- Implementierung von **Auswählen**

```
static public void Auswählen(CDatenSatz[] Feld) {
  for (int anfang = 1; anfang < Feld.length; anfang++) {
    // 1:: anfang = 1, 2, .. , length-1
    int minimum = anfang-1;
    // 2:: das erste Element ist zunächst Minimum
    for (int index = anfang;
             index < Feld.length; index++) {
      // 3:: index = anfang, anfang+1, .. length-1
      if (Feld[minimum].key > Feld[index].key) {
      // 4:: index weist auf kleineres Element
          minimum = index;
    } }
    CDatenSatz swap = Feld[minimum];// vertausche kleinstes
    Feld[minimum] = Feld[anfang-1]; // Element an [anfang-1]
    Feld[anfang-1] = swap;
  } }
```

Sortieren durch Auswählen

- Aus allen Elementen des Feldes das kleinste aussuchen
- vertauschen mit erstem
- Von restlichen kleinstes an Anfang. usw.
- Aufwand Vergleiche =
  $N-1+N-2+ \ldots +2+1 = N \cdot (N-1)/2 = \frac{1}{2} \cdot O(N^2)$
- Aufwand Vertauschungen = $N-1$
- Verfahren ist nicht stabil!



Direktes Auswählen – 2

- Implementierung von **Auswählen**
  - Abwärts sortieren

```
static public void AuswählenAbwärts(CDatenSatz[] Feld) {
  for (int ende = Feld.length - 1; ende >= 1; ende--) {
    // 1:: ende = length-1, .. ,2, 1
    int maximum = 0;
    // 2:: das erste Element ist zunächst Minimum
    for (int index = 1;
             index < ende; index++) {
      // 3:: index = 1, 2, .. ende
      if (Feld[maximum].key <= Feld[index].key) {
      // 4:: index weist auf größeres Element
          maximum = index;
    } }
    CDatenSatz swap = Feld[maximum];// vertausche größtes
    Feld[maximum] = Feld[ende];     // Element an [ende]
    Feld[ende] = swap;
  } }
```

Direktes Auswählen – 3

- Implementierung von **Auswählen**
  - Gleichzeitig kleinstes und größtes Element auswählen

```
static public void auswählenMinMax(CDatenSatz[] Feld) {
  int oben = Feld.length - 1; // 1:: oberer Index
    for (int unten = 0; unten < oben; unten++) {
    //  2:: unten = 0,1,..,oben-1,oben = length-1, length-2,..,unten+1
      int minimum = unten;
      int maximum = unten;
      // 4:: min, max stehen an index=unten
      for (int index = unten + 1;
               index <= oben; index++)
        // 5:: index = anfang+1,..,oben
        if (Feld[minimum].key > Feld[index].key)
                                // 6:: index weist auf
          minimum = index;      //     kleineres Element
        else if (Feld[maximum].key <= Feld[index].key)
                                // 7:: index weist auf
          maximum = index;  //     größeres Element
```

Direktes Auswählen – 3 (forts.)

- Implementierung von **Auswählen**
  - Gleichzeitig kleinstes und größtes Element sortieren

```
    ...
    CDatenSatz swap = Feld[minimum];// 8:: vertausche
    Feld[minimum] = Feld[unten]; // minimum mit unten
    Feld[unten] = swap;
    swap = Feld[maximum];   // 8:: vertausche maximum
    Feld[maximum] = Feld[oben]; // mit oben
    Feld[oben] = swap;
    oben--;                 // 9:: dekrementiere oben
} }
```

Direktes Auswählen – 3 (forts.)

- Implementierung von **Auswählen**
  - Gleichzeitig kleinstes und größtes Element sortieren

```
    ...
    CDatenSatz swap = Feld[minimum];// 8:: vertausche
    Feld[minimum] = Feld[unten]; // minimum mit unten
    Feld[unten] = swap;
    if (maximum == unten)
      maximum = minimum;
    swap = Feld[maximum];   // 8:: vertausche maximum
    Feld[maximum] = Feld[oben]; // mit oben
    Feld[oben] = swap;
    oben--;                 // 9:: dekrementiere oben
} }
```

Direktes Auswählen – 4

- Implementierung **Auswählen 2 Kleinste**
  - Gleichzeitig die beiden kleinsten Elemente sortieren
  - ```
    public void auswählen2Mins(CDatenSatz[] Feld) {
      int minimum;
      int minimum2;
      for (int anfang = 1; anfang < Feld.length;
               anfang+=2) {
        minimum = anfang - 1;
        minimum2 = anfang;
        for (int index = anfang; index < Feld.length;
                 index++)
          if (Feld[minimum2].key >= Feld[index].key)
            if (Feld[minimum].key > Feld[index].key){
              // minimum ist kleiner
              minimum2 = minimum;
              minimum = index;
            } else minimum2 = index; // setzt 2. Mini
    ```
    ```

Direktes Auswählen – 4

- Implementierung **Auswählen 2 Kleinste**
  - Gleichzeitig die beiden kleinsten Elemente sortieren - 2
  - ```
        ...
        CDatenSatz swap = Feld[minimum];
        // vertausche erstes Minimum
        Feld[minimum] = Feld[anfang - 1];
        Feld[anfang - 1] = swap;
        if(minimum2==anfang-1)  // 2.Mini auf Ziel
          minimum2 = minimum;   // setze neuen Wert
        swap = Feld[minimum2];  // vertausche 2. Min
        Feld[minimum2] = Feld[anfang];
        Feld[anfang] = swap;
    } }
    ```
    ```

Direktes Auswählen – 5

- **Auswählen N Kleinste**
  - Gleichzeitig die N kleinsten Elemente sortieren
  - Legt Referenzen auf kleinste Elemente in Liste ab
  - Die Liste ist sortiert
  - Fügt die Elemente entsprechend ihrer Sortierung ein
  - Problem: kleinstes Elemente an Einfügestelle!

Direktes Auswählen – 5

- Implementierung **Auswählen N Kleinste**

```
Void auswählen3(CDatenSatz[] Feld, int anzahl) {
  int oben = Feld.length - 1;  // obere FeldGrenze
  if(anzahl<=1) return;
  int liste [] = new int[anzahl]; // Liste der Indic
  for (int anfang = 0; anfang < oben; ) { //0,1,..
    int listEnde = 0;  // Indices in Liste
    liste[listEnde++] = anfang; // erster Index
    for (int index = anfang + 1; index <= oben;
              index++) { // suche kl. Element
      if (listEnde < liste.length) { // fülle Liste
        listEnde++;
        int listIndex = listEnde - 2;
        for (; listIndex >= 0; listIndex--)
          if (Feld[liste[listIndex]].key <
                                    Feld[index].key)
            liste[listIndex + 1] = liste[listIndex];
          else break;
        liste[listIndex + 1] = index;
      } else  // Liste ist aufgefüllt
```

Direktes Auswählen – 5

- Implementierung **Auswählen N Kleinste**

```
        } else  // Liste ist aufgefüllt
          if (Feld[liste[0]].key > Feld[index].key){
            int listIndex = 1; // finde Einfügestelle
            for ( ; listIndex < listEnde; listIndex++)
              if (Feld[liste[listIndex]].key >
                                        Feld[index].key)
                liste[listIndex-1] = liste[listIndex];
              else break;
            liste[listIndex-1] = index;
          } }
        // tausche Elemente an richtigen Platz (anfang)
         for(int listIndex=listEnde-1;listIndex>=0;
```

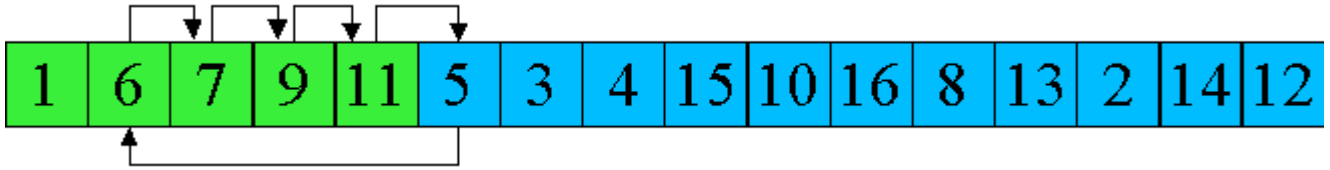
Direktes Auswählen – 5

- Implementierung **Auswählen N Kleinste**

```
        // tausche Elemente an richtigen Platz (anfang)
        for(int listIndex=listEnde-1;listIndex>=0;
          /*gehe Liste durch*/          listIndex--) {
          int von = liste[listIndex];
          // Element soll in 'von' stehen
          while(von<anfang) // solange Element
                von = liste[listIndex+(anfang-von)];
          if(von!=anfang) { // vertausche Element
            CDatenSatz swap = Feld[von];
            Feld[von] = Feld[anfang];
            Feld[anfang] = swap;
          }
          anfang++;// erhöhe Einfügestelle
    } } }
```

Sortieren durch Einfügen

- sucht zu jedem Element den richtigen Platz
- Verschiebe erstes unsortierte Element nach links, bis das linke Element kleiner oder gleich diesem
- Aufwand Vergleiche $\approx ½ + 2/2 + 3/2 + .. (N-1)/2 = N\cdot(N-1)/4 = 1/4\cdot O(N^2)$
- Aufwand Verschiebungen $\approx N-1 + N\cdot(N-1)/4 = O(N)+1/4\cdot O(N^2)$



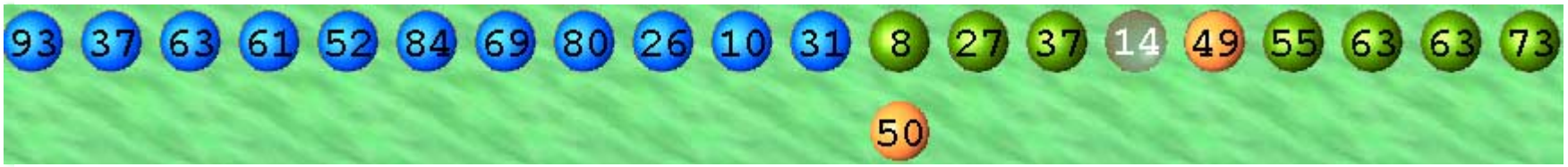
Sortieren durch Einfügen

- Implementierung **Direktes Einfügen**
  - Einfügen aufwärts


```
static public void einfügen(CDatenSatz[] Feld) {
   if (Feld == null) return;    // kein Feld
   if (Feld.length <= 1) return;// Feld immer sortiert
   for (int anfang = 1; anfang < Feld.length;
         anfang++) {// Feld[0..Anfang-1] ist
     CDatenSatz aktuell = Feld[anfang];
     int index = anfang - 1;
     for ( ; index >= 0; index--) {
       if (Feld[index].key <= aktuell.key) break;
       Feld[index + 1] = Feld[index];
     }
     Feld[index + 1] = aktuell;
   } }
```

Sortieren durch Einfügen

- Implementierung **Direktes Einfügen**
  - Einfügen abwärts




```
static public void einfügenAb(CDatenSatz[] Feld) {
  if (Feld == null) return;    // kein Feld
  if (Feld.length <= 1) return;// Feld immer sortiert
  for (int anfang = Feld.length; anfang >= 0;
        anfang--) {// Feld[0..Anfang-1] ist
    CDatenSatz aktuelles = Feld[anfang];
    int index = anfang + 1;
    for ( ; index < Feld.length; index++) {
      if (Feld[index].key >= aktuelles.key) break;
      Feld[index - 1] = Feld[index];
    }
    Feld[index - 1] = aktuelles;
  } }
```

Sortieren durch Einfügen

- Implementierung **Direktes Einfügen**
  - Einfügen in einen Abschnitt des Feldes
  - ```
    static public void einfügen(CDatenSatz[] Feld,
                                 int von, int bis) {
      if (von<0) return;                  // unzulässiger Index
      if (Feld.length < bis) return;// Index??
      if (Feld.length <= 1) return; // sortiert
      if (von==bis) return;               // sortiert
      for (int anfang = von + 1; anfang<=bis; anfang++){
        // Feld[von .. anfang-1] ist sortiert
        CDatenSatz aktuelles = Feld[anfang];
        int key = aktuelles.key;
        int index = anfang - 1;
        for (  ; index >= von; index--) { // sort.Teilflg
          if (Feld[index].key <= key) break;// vergleiche
          Feld[index + 1] = Feld[index];    // verschiebe
        }
        Feld[index + 1] = aktuelles;// kopiere Aktuelles
    } }
    ```
    ```

Sortieren durch Einfügen

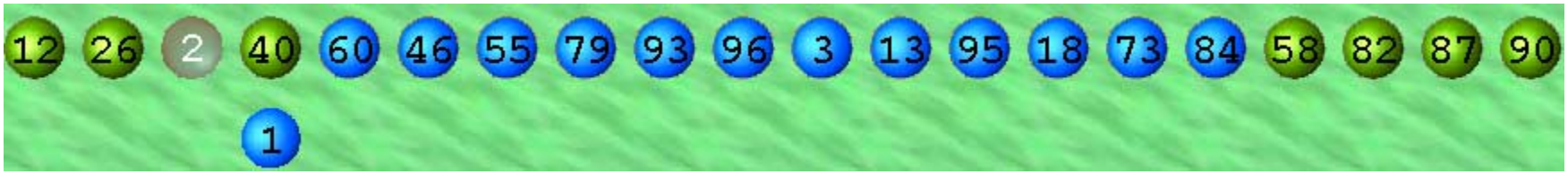
- Einfügen alternierend – 1
  - Füge kleinere Elemente unten, größere oben ein.
    - Einfügen von kleinen Werten unten

    - Tausches großes Element mit rechtem Randelement

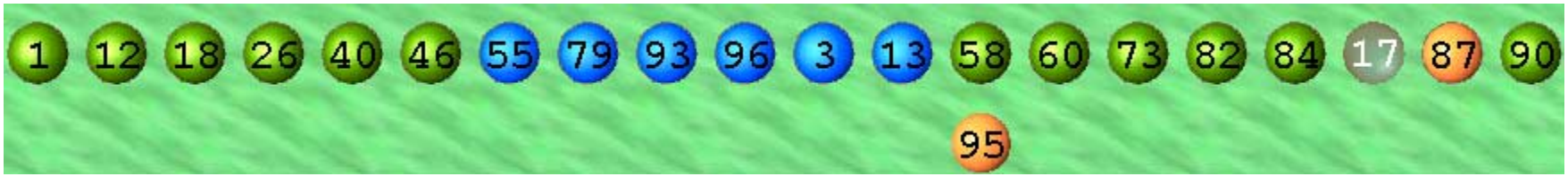
    - Weiteres Einfügen links, große Elemente rechts, usw.

Sortieren durch Einfügen

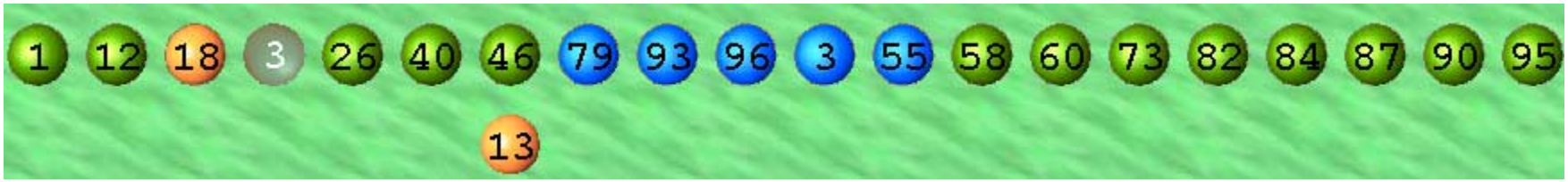
- Einfügen alternierend - 2
  - Füge kleinere Elemente unten, größere oben ein.
    - Einfügen von kleinen Werten unten



    - Einfügen von großen Werten oben



    - Vertauschen der Randelemente und weiter Einfügen usw.



Sortieren durch Einfügen

- Mehrere Elemente gleichzeitig einfügen
  - Einsparungen durch
    - Größere Verschiebeschritte
    - Geringere Anzahl von Vergleichen
  - Zwei Elemente gleichzeitig einfügen

  - Drei Elemente gleichzeitig einfügen

  - Parametrisierte Anzahl (=5) von Elementen einfügen

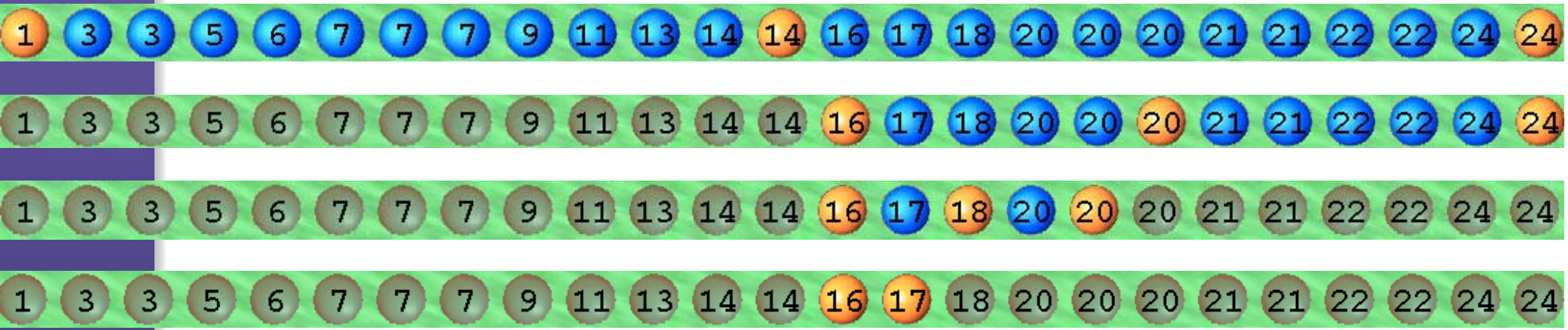
Sortieren durch Einfügen

- Mehrere Elemente gleichzeitig einfügen - 2
  - Weitere Einsparungen durch
    - Setze 1. Element auf Pivot-Element
    - Vergleiche kleinstes aus Liste mit 1. Element, vertausche ggf.
    - Optimiere Länge der Einfügeliste
      ein Optimum liegt bei Quadratwurzel aus Anzahl
  - Einige dieser Ansätze können Stabilität verletzen
  - In der Regel bleibt jedoch Ordnungsverträglichkeit erhalten

Binäres Suchen

- Lineares Suchen
  - Gehe alle Elemente sequentiell durch

- Binäres Suchen
  - Teile Suchintervall in zwei Hälften, z.B. suche 17



Binäres Suchen

- Implementierung Binäres Suchen

```
static int binäresSuchen(CDatenSatz[] Feld,int key){
  int von = 0;                    // linker Index
  int bis = Feld.length - 1;      // rechter Index
  int mitte = 0;                  // mittlerer Index
  while (true) { // Ende falls gef. oder Intervall=0
    mitte = (von + bis) / 2;      // mittleres Element
    if (key < Feld[mitte].key) // Schlüssel links
      bis = mitte - 1;
    else if (key > Feld[mitte].key)//Schlüssel rechts
      von = mitte + 1;
    else
      return mitte;               // Schlüssel gefunden
    if (von > bis)                // Schlüssel nicht da
      return -1;                  // ... ergibt -1
} }
```

Sortieren durch Einfügen

- Suche Einfügestelle durch Binäres Suchen
  - Schiebe größere Elemente ohne Wertevergleich nach rechts

  - Dieses lässt sich auch Kombinieren
    - zwei Elemente gleichzeitig einfügen

    - drei Elemente gleichzeitig einfügen

Shellsort

- Idee: Einfügen über größere Distanzen
  - (D.L. Shell 1959)
  - Analyse bis heute nicht vollständig möglich
- Vorteile
  - Einfach zu implementieren
  - Laufzeitkomplexität im Normalfall $O(N^{3/2})$
  - Verbesserte Verfahren (empirisch) von Sedgewick
    - z.B. Laufzeitkomplexität im Normalfall $O(N^{7/6})$
- Shellsort wird von vielen Autoren empfohlen!

Shellsort

- Beispiel für Shell Sort
  - Im Abstand der Werte ..., 13, 4, 1 Direktes Einfügen

Shellsort – Algorithmus

- Algorithmus zu Shellsort

```
static public void einfügen(CDatenSatz[] Feld) {
  int abstand = 1;
  while (abstand <= Feld.length)
         abstand = 3 * abstand+1;
  while (abstand >= 3) { //abstand=...,40,13,4, 1
    abstand /= 3;          // nächster Abstand
    for (int oben = abstand; oben < Feld.length;
             oben++) { // Einfügen in Abstand
      CDatenSatz swap = Feld[oben];//Einzufügendes
      int vergleich = oben - abstand;
      for (; vergleich >= 0; vergleich -= abstand)
        // finde kleineres Element in -n*abstand
        if (Feld[vergleich].key > swap.key)
          // falls größer, schiebe hierher
          Feld[vergleich+abstand]=Feld[vergleich];
        else break;              // sonst höre auf
      Feld[vergleich+abstand] = swap;// einfügen
} } }
```

Shellsort – Algorithmus

- Andere Folge von Werten nach Sedgewick
  - Werden die Folgenwerte
    - 1,5,19,41,109, ...
  - verwendet, so verbessert sich der Algorithmus um etwa 10% gegenüber dem Algorithmus mit der Folge 1, 4, 13, 40, ... Die Folgenglieder lassen sich aus den Formeln

    $$n_{2i+1} = n_{1,3,5,\ldots} = 1 + 9 \cdot 4^i - 9 \cdot 2^i$$

    oder

    $$n_{2 \cdot i - 2} = n_{2,4,6,\ldots} = 1 + 4^i - 3 \cdot 2^i$$

    bestimmen.

Shellsort – Algorithmus

- Algorithmus zu Shellsort mit Sedgewick-Folge

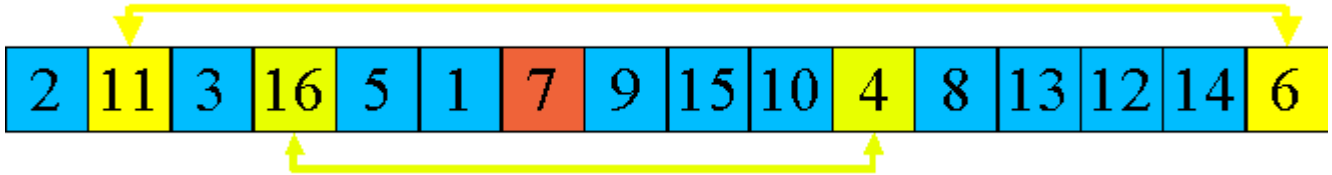
```
static public void einfügen(CDatenSatz[] Feld) {
  int[] abstände = {1,5,19,41,109,209,505,929,
    2161,3905,8929,16001,36289,64769,146305,
    260609,587521,1045505,2354689,4188161,
    9427969, 16764929, 37730305, 67084289};
  int index=0;
  while (index<abstände.length &&
                abstände[++index]<Feld.length);
    index--;
    int abstand = 1;
    while (index >= 0) {
      abstand = abstände[index--];
      for (int oben = abstand; oben < Feld.length;
                oben++) {
        CdatenSatz swap = Feld[oben];
        int vergleich = oben - abstand;
        ...
} } }
```

Shellsort – Algorithmus

- Algorithmus zu Shellsort mit Sedgewick-Folge
  -
```
static public void einfügen(CDatenSatz[] Feld) {
        ...
        for (int oben = abstand; oben < Feld.length;
                oben++) {
          CDatenSatz swap = Feld[oben];
          int vergleich = oben - abstand;
          for (;vergleich>=0; vergleich-=abstand)
            if (Feld[vergleich].key > swap.key)
              Feld[vergleich + abstand] =
                                    Feld[vergleich];
            else break;
          Feld[vergleich + abstand] = swap;
  } } }
```

Quicksort

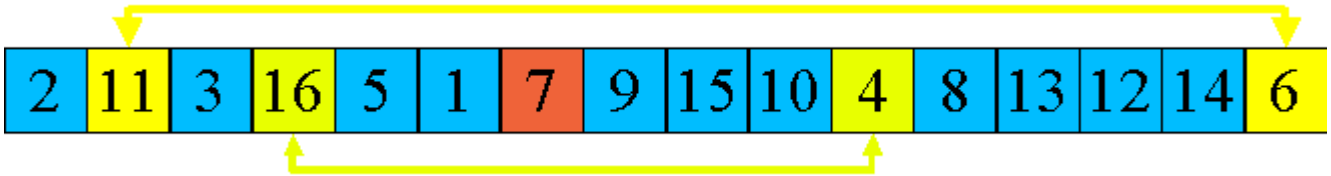
- Idee
  - Verteile Daten auf zwei Teilmengen
    - linke Feldhälfte 'kleine' Schlüssel
    - rechte Feldhäfte 'große' Schlüssel.
    - kleiner Schlüssel =< Vergleichselement
    - großer Schlüssel >= Vergleichselement
    - für beide Feldhälften fortgesetzt, bis alle Elemente sortiert



  - Implementierung meist rekursiv
  - Auch nicht rekursive Implementierung möglich

Quicksort

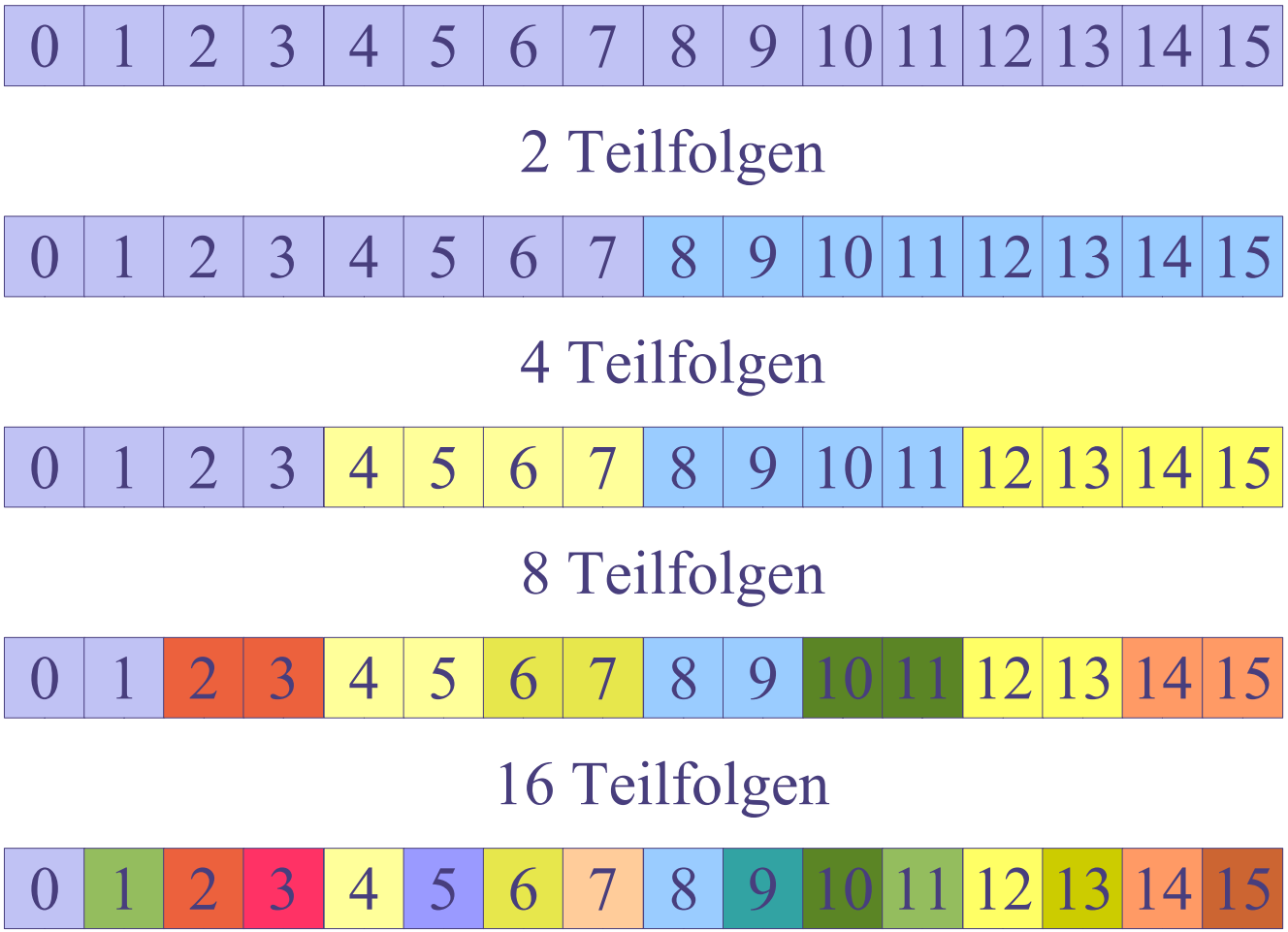
- Algorithmus
  - Abfrage der Feldlänge (>1)
  - wähle Vergleichselement
    - möglichst „in der Mitte“ – Median
    - nur ein Element – quadratische Laufzeit
  - Schleife



    - identifiziere erstes links stehende, größere Element,
    - identifiziere erstes rechts stehende, kleinere Element,
    - vertausche beide Elemente
  - bei nächsten Elementen fortsetzen
    - Wenn linke Grenze `indexL` rechte `indexR` überholt
      → Teilfelder rekursiv weiter sortieren

Quicksort

- Aufwand: $N \cdot log_2 N$

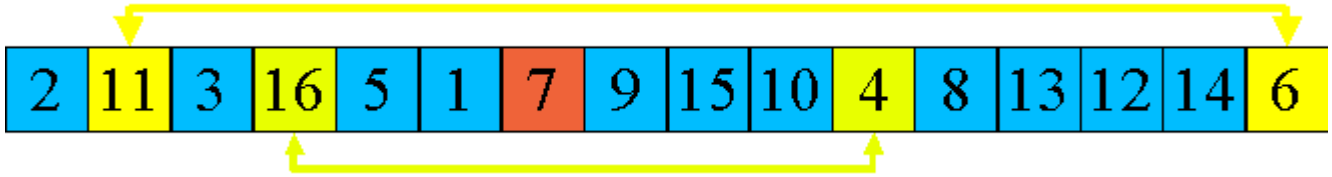


Quicksort

- Quicksort sehr schneller Algorithmus
  - Laufzeit in extremen Situationen quadratisch
  - zusätzlicher Speicherplatz benötigt (Laufzeitstack)
    - relativ viele Datensätze
- Andere Algorithmen sicherer
  - z.B. Heapsort,
  - im Mittel etwa um ein Drittel langsamer
- Andere Algorithmen schneller
  - Distribution-Sort
  - aufwendiger zu implementieren
  - Zusätzlicher Speicherplatz (Quadratwurzel)

Quicksort

```
void quick(CDatenSatz[] Feld, int Links, int Rechts) {
     if (Links < Rechts) {
       int indexL = Links, indexR = Rechts;
       int vergleich = median3(Feld[indexL].key,
         Feld[indexR].key, Feld[(indexL+indexR)/2].key);
         while (indexL <= indexR) {
           while (vergleich > Feld[indexL].key) indexL++;
           while (vergleich < Feld[indexR].key) indexR--;
           if(indexL <= indexR) {
             CDatenSatz swap = Feld[indexL];
             Feld[indexL] = Feld[indexR];
             Feld[indexR] = swap;
             indexL++;
             indexR--;
         } }
         quick(Feld, Links, indexR);
         quick(Feld, indexL, Rechts);
     } }
```



Quicksort

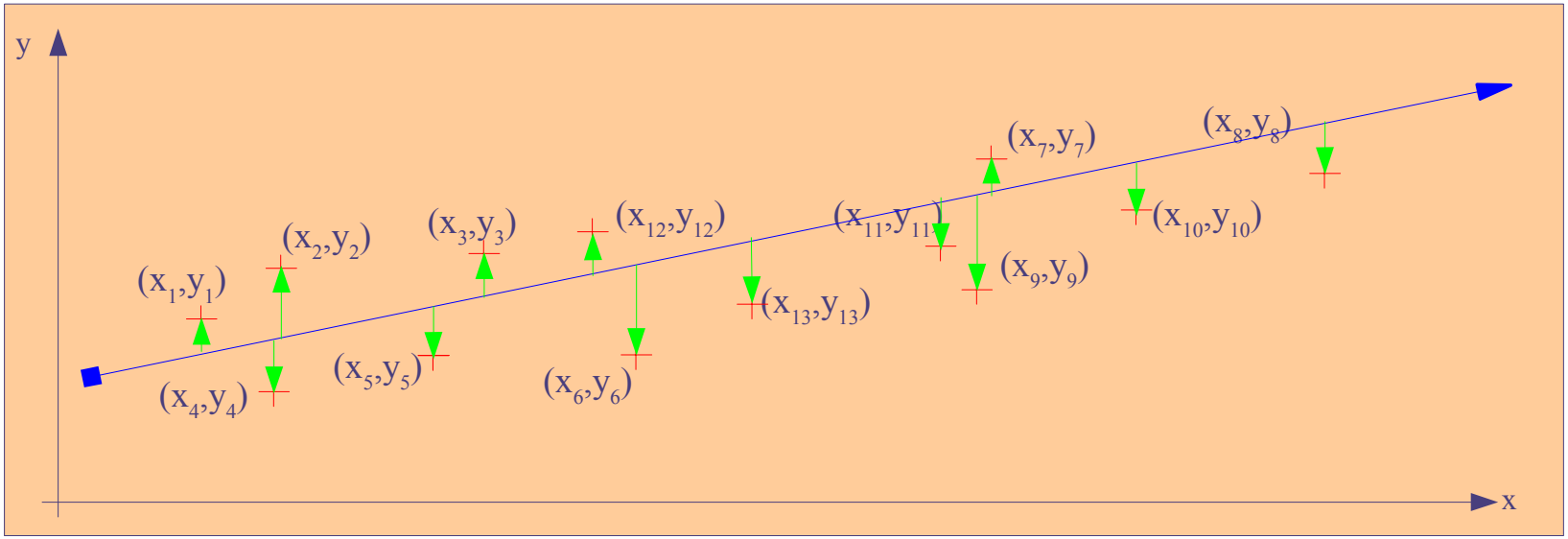
```
int median3(int a, int b, int c) {
  if(a<b) {
    if(b<c) return b;    // a<b<c
    if(c<a) return a;    // c<a<b
    return c;            // a<=c<=b
  } else {               // a>=b
    if(a<c) return a;    // b<=a<c
    if(c<b) return b;    // c<b<=a
    return c;            // b<=c<=a
  }
}
```

Lineare Regression

- Menge von Punkten (x_i,y_i).
  - Gesucht: Gerade $y(x) = a+b\cdot x$, so dass

$$Q(a,b)=\sum_{i=0}^{n}\left(y_i-y(x_i)\right)^2=\sum_{i=0}^{n}\left(y_i-a-b\cdot x_i\right)^2$$

minimal wird, d.h. der mittlere Abstand zu allen Punkten am kleinsten



Lineare Regression

- Menge von Punkten (x_i, y_i).
  - Gesucht: Gerade $y(x) = a + b \cdot x$, so dass

$$Q(a,b) = \sum_{i=0}^{n} \left(y_i - y(x_i) \right)^2 = \sum_{i=0}^{n} \left(y_i - a - b \cdot x_i \right)^2$$

minimal wird, d.h. der mittlere Abstand zu allen Punkten am kleinsten.

- Ausmultiplizieren $Q(a,b) = \sum_{i=0}^{n} \left(y_i^2 + a^2 + b^2 \cdot x_i^2 - 2 \cdot y_i \cdot a - 2 \cdot y_i \cdot b \cdot x_i + 2 \cdot x_i \cdot a \cdot b \right)$
- Ableitung nach a $Q^a(b) = \sum_{i=0}^{n} \left(2 \cdot a - 2 \cdot y_i + 2 \cdot x_i \cdot b \right) = 0$
- Ableitung nach b $Q^b(a) = \sum_{i=0}^{n} \left(2 \cdot b \cdot x_i^2 - 2 \cdot y_i \cdot x_i + 2 \cdot x_i \cdot a \right) = 0$
- Abkürzungen $X = \frac{1}{n} \cdot \sum_{i=0}^{n} x_i, \; Y = \frac{1}{n} \cdot \sum_{i=0}^{n} y_i, \; Z_{11} = \frac{1}{n} \cdot \sum_{i=0}^{n} x_i \cdot y_i, \; X_2 = \frac{1}{n} \cdot \sum_{i=0}^{n} x_i^2$

Lineare Regression

- Lösung des Gleichungssystems

$$a - Y + X \cdot b = 0$$
$$b \cdot X_2 - Z_{11} + X \cdot a =$$
$$b \cdot X_2 - Z_{11} + X \cdot (Y - X \cdot b) =$$
$$b \cdot X_2 - Z_{11} + X \cdot Y - X^2 \cdot b = 0$$

$$b = \frac{Z_{11} - X \cdot Y}{X_2 - X^2} = \frac{E[XY] - E[X] \cdot E[Y]}{E[X^2] - E[X]^2} = \frac{cov(X, Y)}{Var(X)}$$

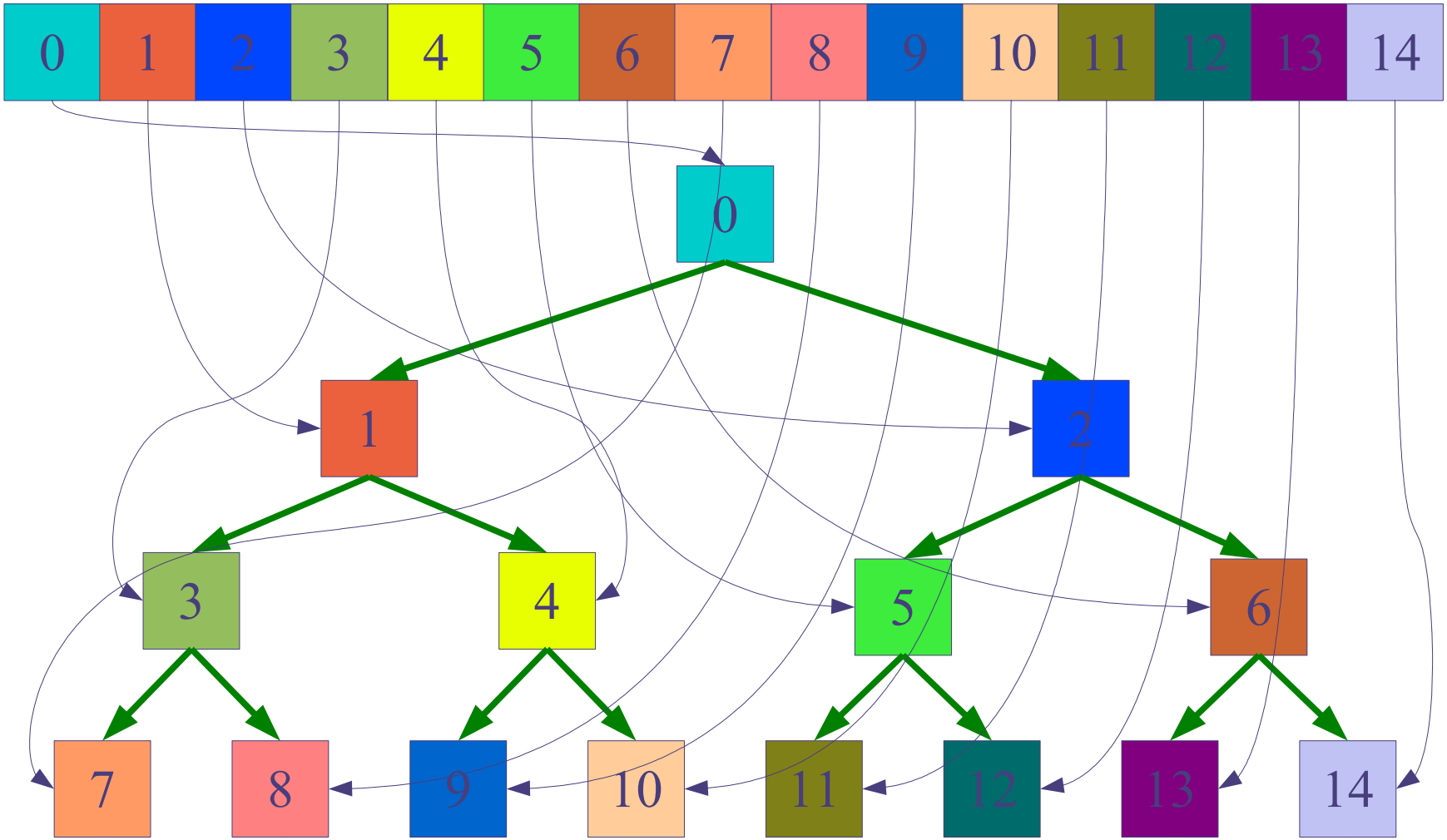
$$a = Y - X \cdot b = Y - X \cdot \frac{Z_{11} - X \cdot Y}{X_2 - X^2} = E[Y] - E[X] \cdot b$$

Heapsort

- Solides Sortierverfahren
  - Daten werden partiell vorsortiert
  - 'größtes Element' an letzten Platz gebracht
  - Felde verkürzt usw.
- Aufwand
  - Für partielles Sortieren: $O(\log_2 N)$
  - Gesamtaufwand daher: $O(N \cdot \log_2 N)$
- Interessante Datenstruktur: Heap

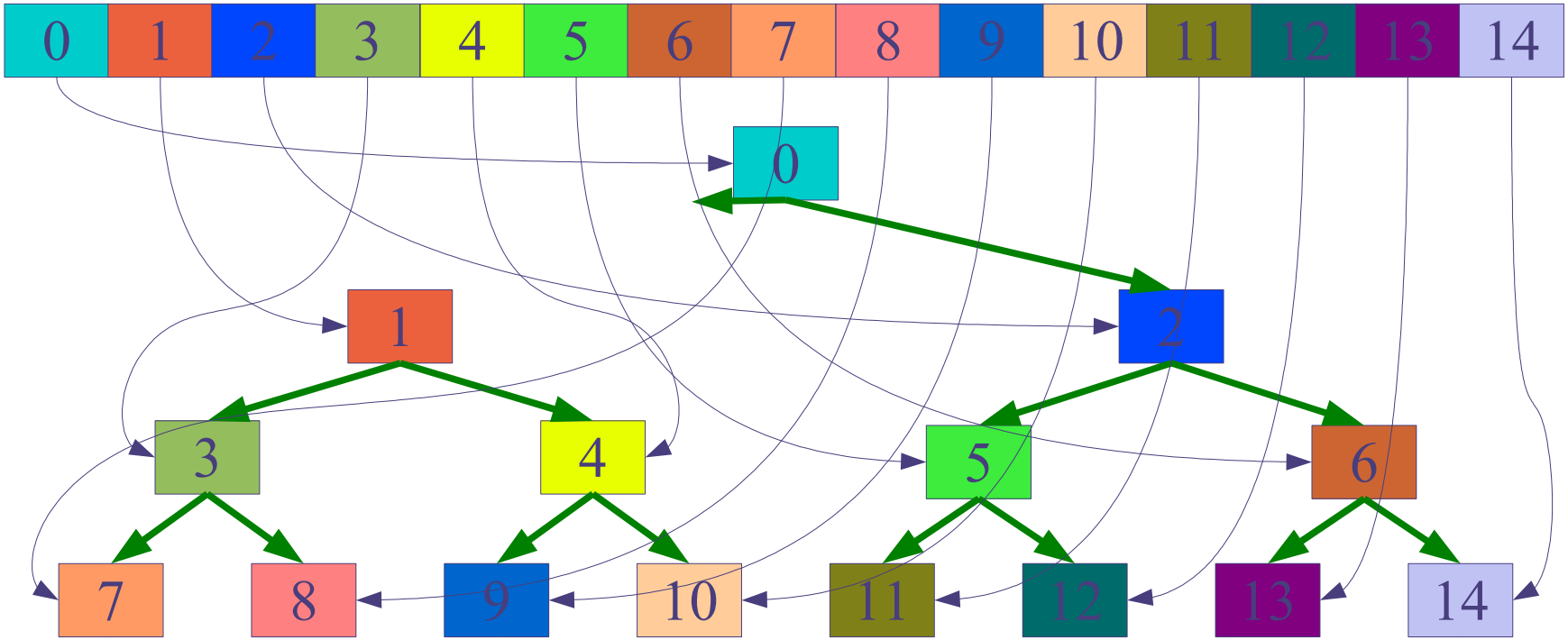
Heapsort

- Baumstruktur auf Feld



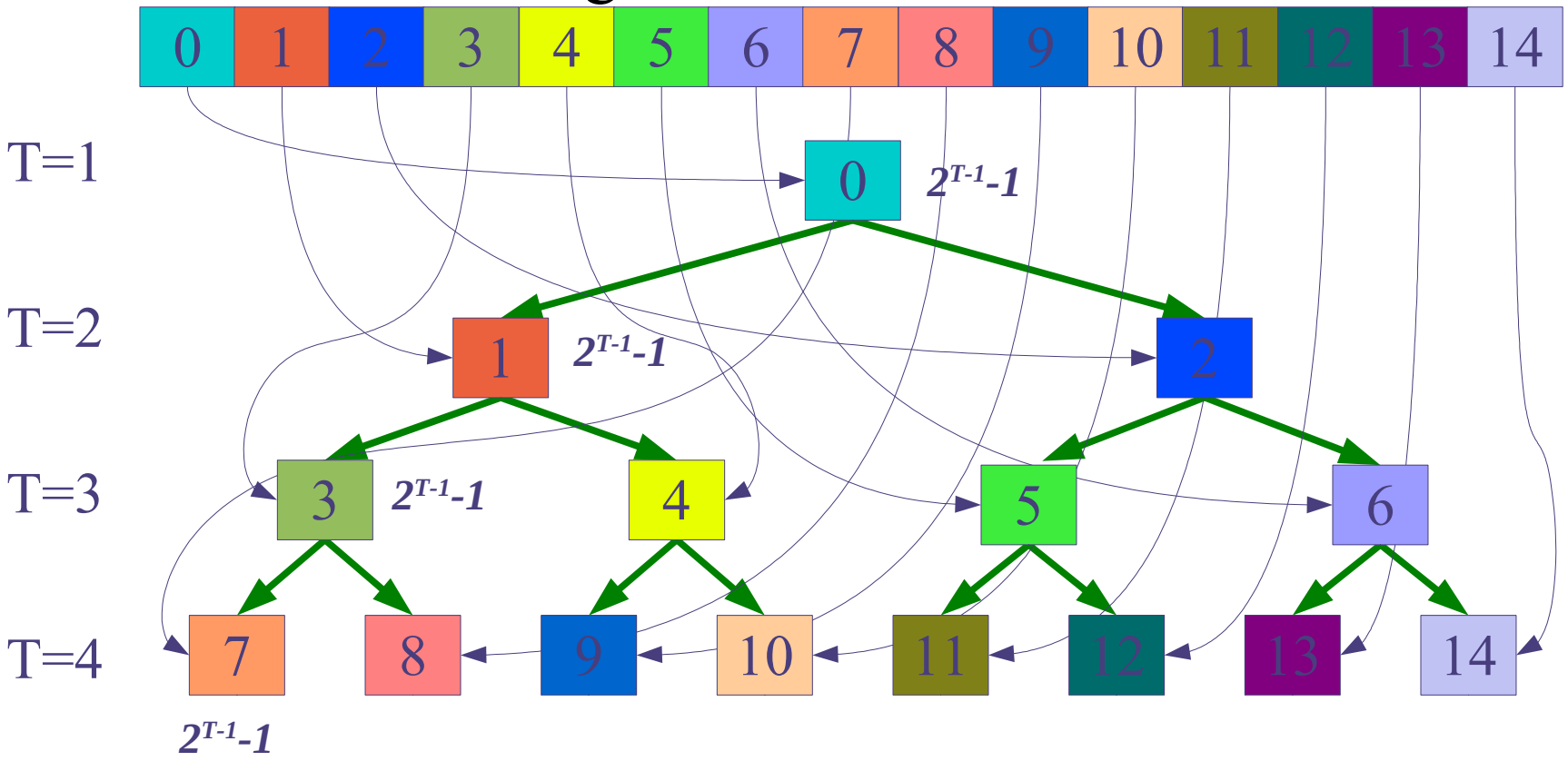
Heapsort

- Baumstruktur auf Feld
  - Linker Nachfolger: 2·k+1 (bei Indizierung ab 0)
  - Rechter Nachfolger: 2·k+2



Heapsort

- Baumstruktur auf Feld
  - Linker Nachfolger: 2·k+1 (bei Indizierung ab 0)
  - Rechter Nachfolger: 2·k+2



Heapsort

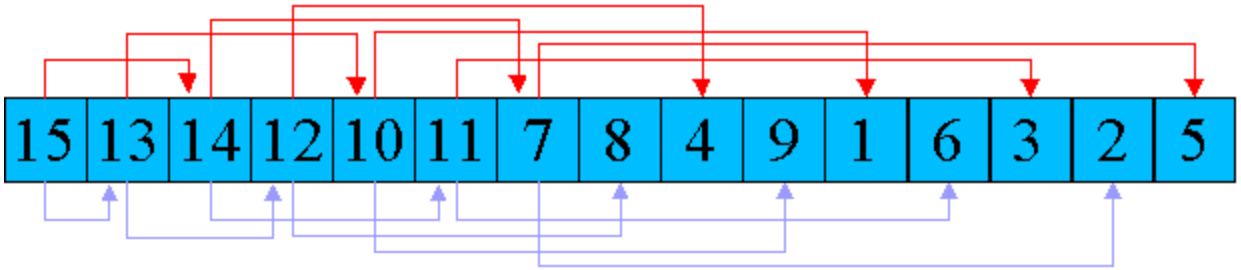
- Anzahl der Knoten in Tiefe $T=2^{T-1}$
  - $2^{T-1}[\text{Anzahl der Knoten in Tiefe T=1=Wurzel}]=2^{1-1}=1$
  - $2^{T-1}[\text{Anzahl der Knoten in Tiefe T}]$
  - $2\cdot(2^{T-1})[\text{Anzahl der Knoten in Tiefe T+1}]=2^{T}$
- Nummer der linken Knoten in einer Zeile $2^{T-1}-1$
  - $2^{1-1}-1[\text{Knotennummer links Tiefe 1=Wurzel}]=0$
  - $2^{T-1}-1[\text{Knotennummer links Tiefe T}]$
  - $2^{T-1}-1[\text{Knotennummer links Tiefe T}]+$
    $2^{T-1}[\text{Anzahl der Knoten Tiefe T}]=$
    $=2^{T}-1[\text{Knotennummer links Tiefe T+1}]$

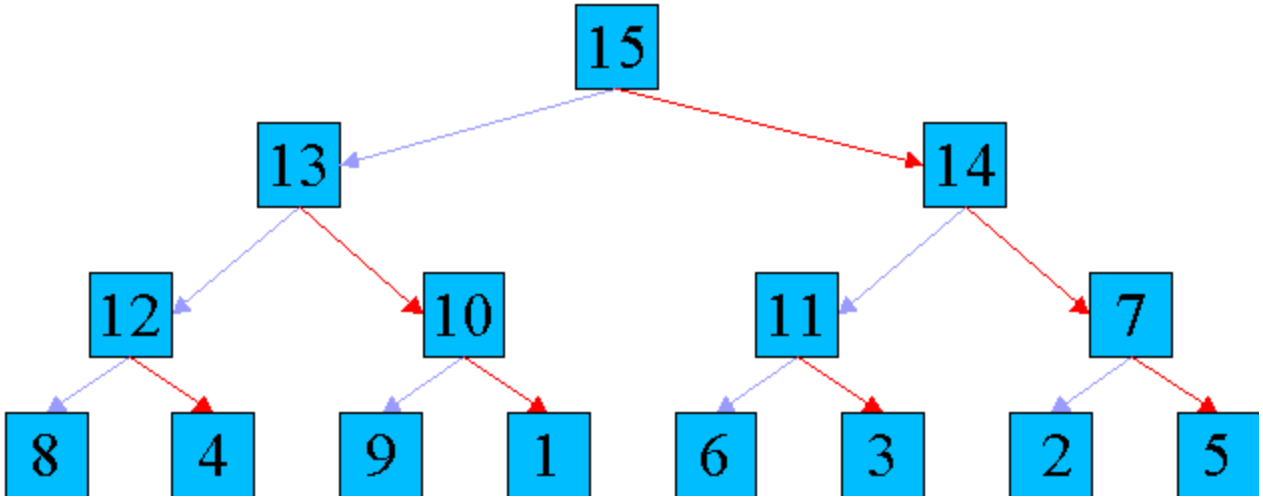
Heapsort

- Nummer der Folgeknoten zu V: $2 \cdot V+1,\ 2 \cdot V+2$
  - $2^{T-1}-1+k$ [Vater-Knotennummer]
  - $2 \cdot (2^{T-1}-1+k)+1$ [linker Sohn] $=(2^{T}-1)+2 \cdot k$
  - $k=0: (2^{T}-1)+2 \cdot 0=2^{T}-1$ [linker Sohn vom linken Knoten]
  - $k=j: (2^{T}-1)+2 \cdot j$ [linker Sohn Knoten $2^{T-1}-1+j$]
  - $k=j+1:$
    $(2^{T}-1)+2 \cdot j+2$ [linker Sohn Knoten $2^{T-1}-1+j+1$] $=$
    $=(2^{T}-1)+2 \cdot (j+1)$
  - $(2^{T}-1)+2 \cdot k+1$ [rechter Sohn] $=2 \cdot (2^{T-1}-1+k)+2$

Heapsort

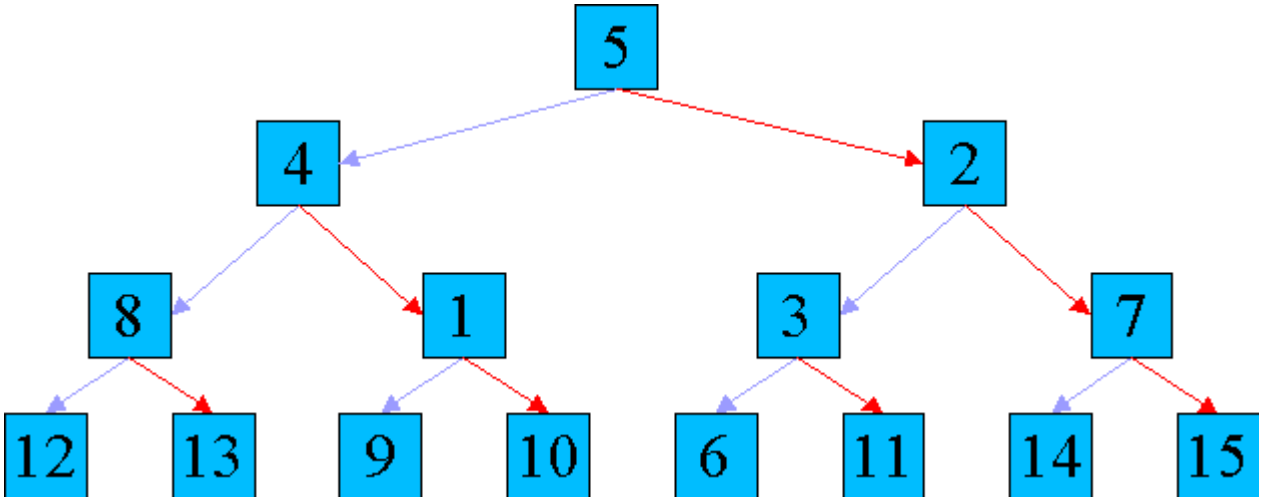
- Heap (mit partieller Ordnung)
  - Feld[k] ≥ Feld[2·k+1]
  - Feld[k] ≥ Feld[2·k+2]





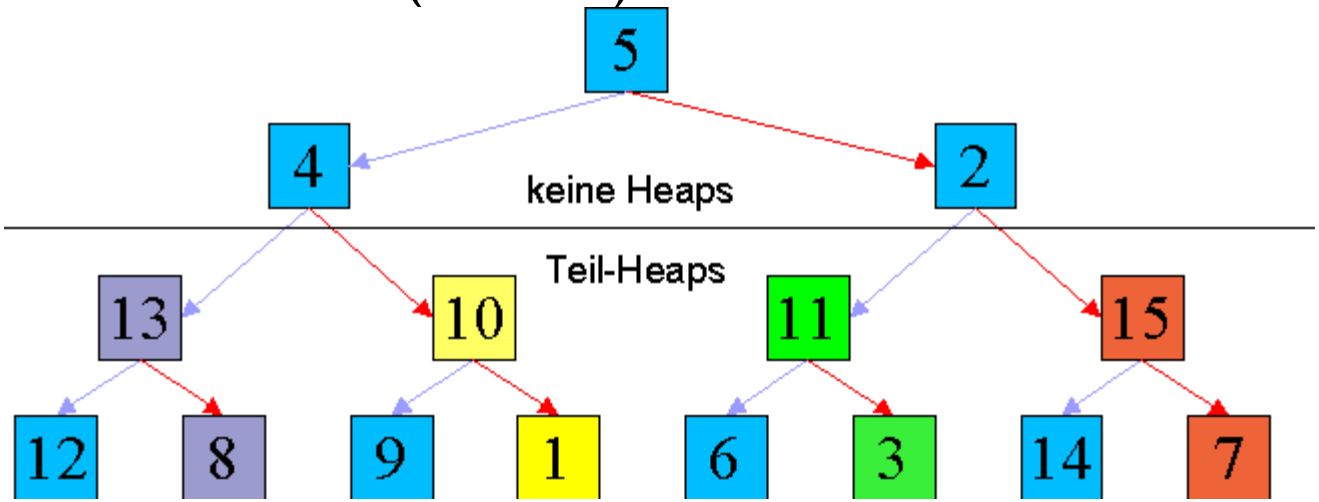
Heapsort

- Aufbau (der partiellen Ordnung) eines Heaps
  - Sämtliche Blätter sind Heaps
  - Aufwand O(0)=0



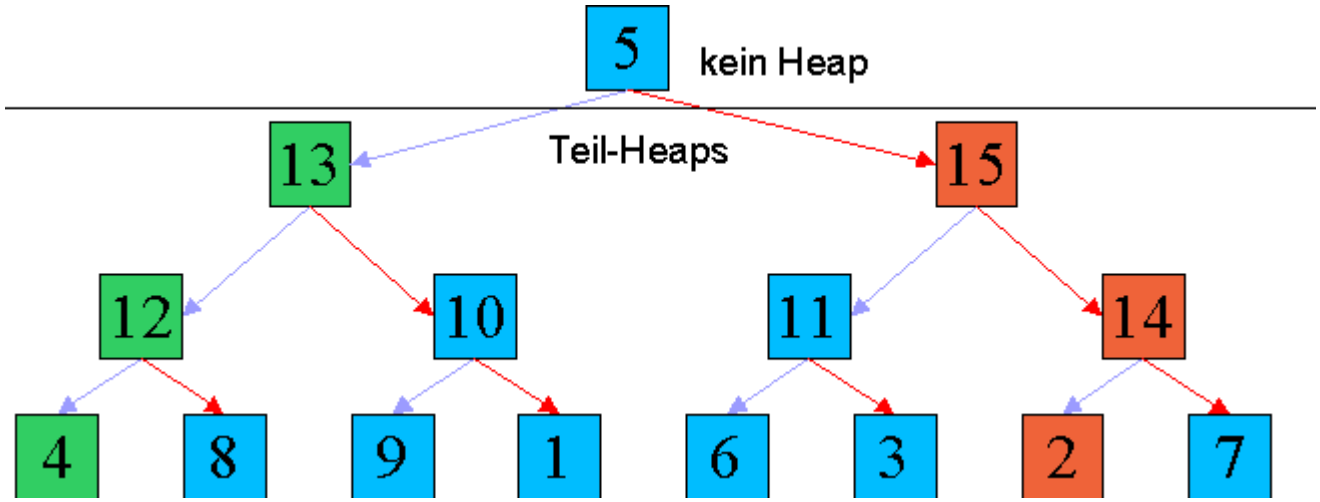
Heapsort

- Aufbau (der partiellen Ordnung) eines Heaps
  - 4 Heaps der Höhe 2, Höhe = T, $N=2^T-1$
  - Aufwand $\approx O(\#\text{Heap-Operationen} \times \#\text{Heaps})$
    $= O(1 \cdot 2^T/4)$



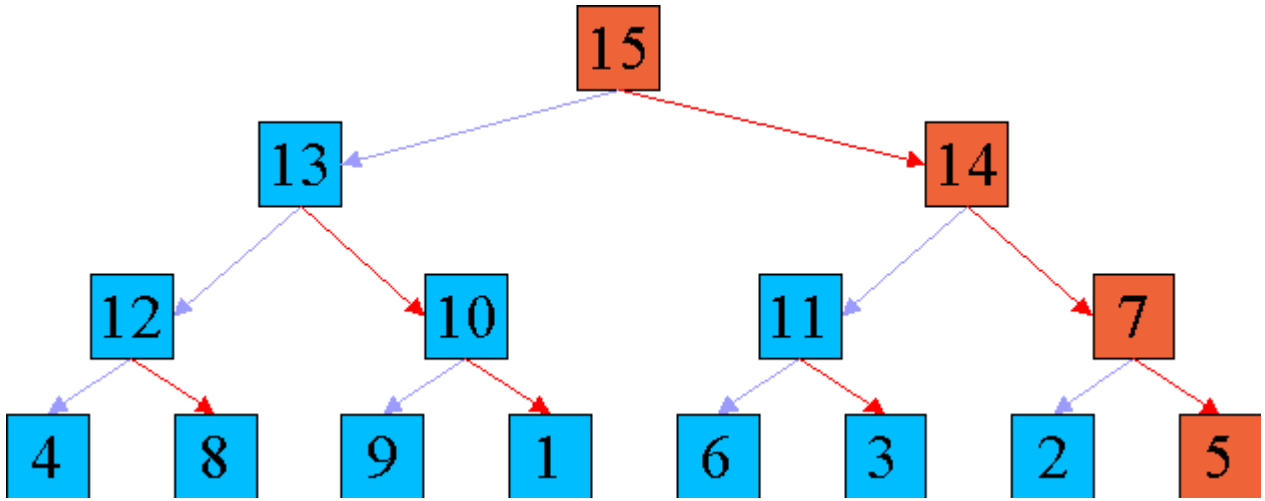
Heapsort

- Aufbau (der partiellen Ordnung) eines Heaps
  - 2 Heaps der Höhe 3, Höhe = T, $N=2^T-1$
  - Aufwand < $O(2 \cdot 2^T/8)$



Heapsort

- Aufbau (der partiellen Ordnung) eines Heaps
  - Heap in voller Höhe = T, $N=2^T-1$
  - Aufwand < $O(3 \cdot 2^T/16)$



Heapsort

- Aufwandsberechnung mit $T = \log_2 N+1$, $N+1 = 2^T$

$$1\cdot\frac{N+1}{4}+2\cdot\frac{N+1}{8}+3\cdot\frac{N+1}{16}+\ldots+(T-1)\cdot\frac{N+1}{2^T}=N-T$$

$$1\cdot 2^{T-2}+2\cdot 2^{T-3}+3\cdot 2^{T-4}+\ldots+(T-2)\cdot 2^1+(T-1)\cdot 2^0=N-T$$

$$T=1: 0\cdot 2^0=0=1-1$$

$$T=2: 0\cdot 2+1\cdot 2^0=0+1=2=3-2$$

$$T=3: 0\cdot 2^2+1\cdot 2^1+2\cdot 2^0=0+2+2=4=7-3$$

$$\sum_{k=1}^{T-1}(T-k)\cdot 2^{k-1}=N-T=2^T-1-T\,.$$

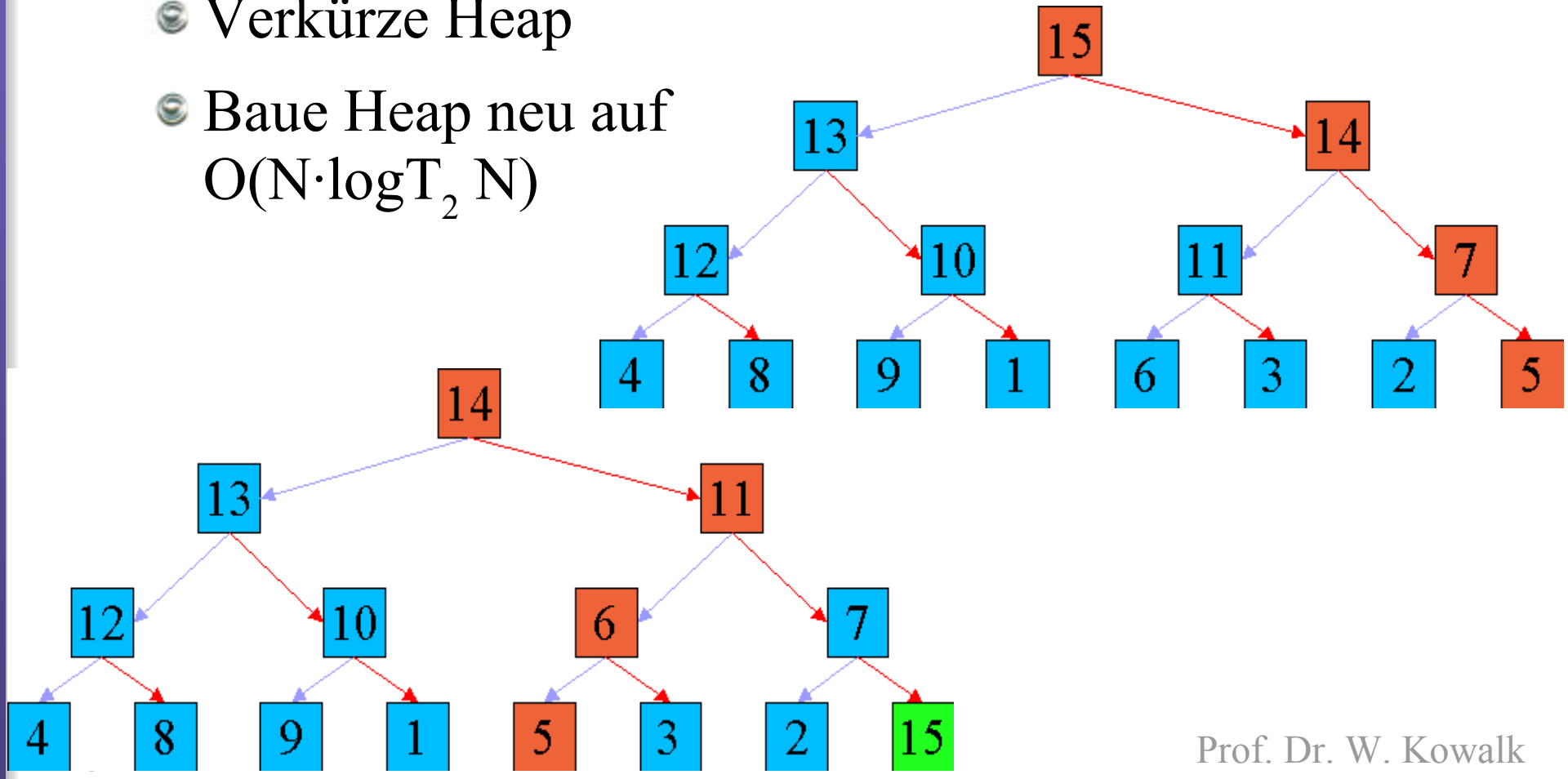
$$\sum_{k=1}^{T}(T+1-k)\cdot 2^{k-1}=\sum_{k=1}^{T-1}(T-k)\cdot 2^{k-1}+\sum_{k=1}^{T-1}2^{k-1}+2^{T-1}=$$

$$=\sum_{k=1}^{T-1}(T-k)\cdot 2^{k-1}+2^{T-1}-1+2^{T-1}=$$

$$=2^T-1-T+2^T-1=2^{T+1}-1-(T+1).$$

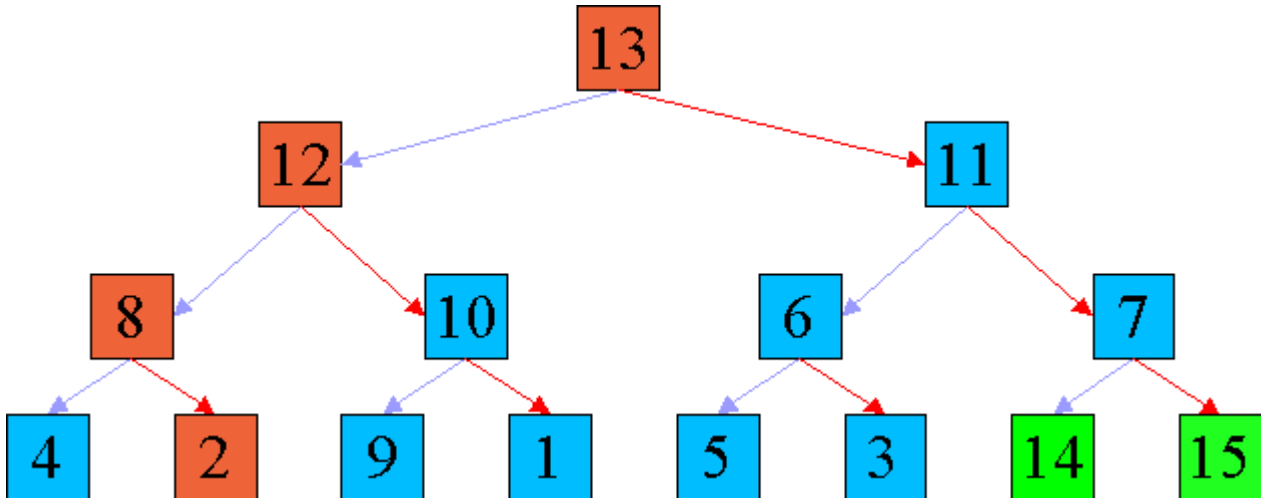
Heapsort

- Sortieren
  - Bringe größtes Element an das Ende des Feldes
  - Verkürze Heap
  - Baue Heap neu auf $O(N \cdot \log T_2 N)$



Heapsort

- Sortieren



- Aufwand für das Sortieren
  - Jedes Element durch ganzen Heap bis Blatt absenken

$$\sum_{k=1}^{T-1} \frac{N+1}{2^k} \cdot (T-k) = (N+1) \cdot \left(T + \frac{1}{2^{T-1}} - 2 \right) \leq (N+1) \cdot T$$

  - $O(N \cdot \log_2 N)$

Heapsort

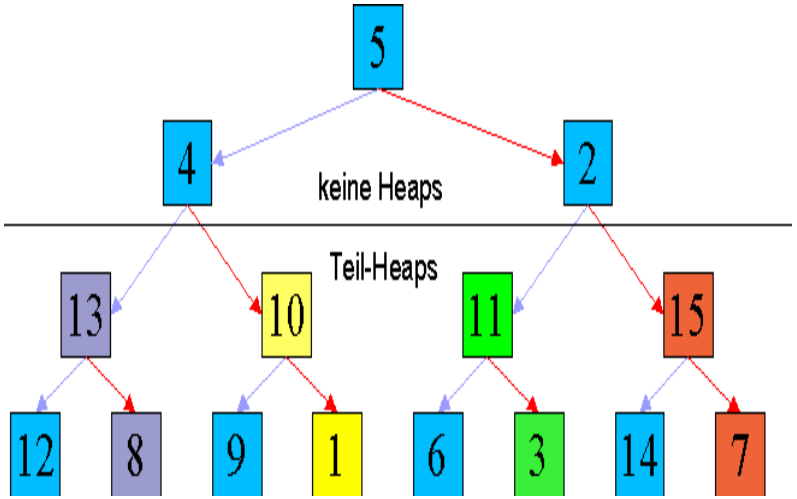
- Interessanter Algorithmus
  - Datenstruktur Baum
    - ohne Referenzen
    - Weniger Speicherplatz, da keine Referenzen zu speichern
  - Vorgänger und Nachfolger leicht ermittelbar
  - Baum leicht horizontal durchlaufbar
  - Datenstruktur nützlich
    - wenn maximale Tiefe eines Baums bekannt

Heapsort

- Heapsort verbessern
  - ternärer oder quarternärer Baum
  - Bottom-Up-Sortieren
- Heapsorts maximale Rechenzeit: O(N·log(N))

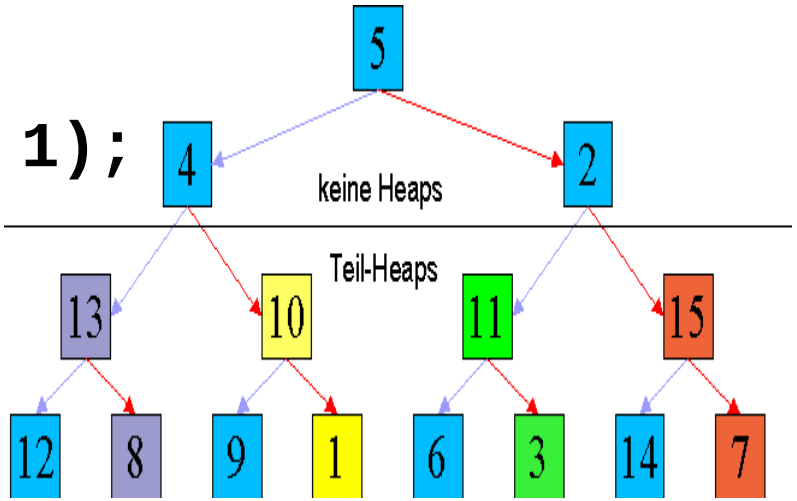
Heapsort

```
void Senke(CDatenSatz[]Feld,int Vater,int Ende){
  CDatenSatz swap = Feld[Vater];
  int Sohn, SohnL, SohnR;
  int key = swap.key;
  while (true) {
    SohnL = (Vater << 1) + 1;
    if (SohnL > Ende) break;
    SohnR = SohnL + 1;
    Sohn = SohnL;
    if (SohnR <= Ende &&
            Feld[SohnL].key < Feld[SohnR].key)
    Sohn = SohnR;
    if (Feld[Sohn].key <= key) break;
      Feld[Vater] = Feld[Sohn];
    Vater = Sohn;
  }
  Feld[Vater] = swap;
}
```



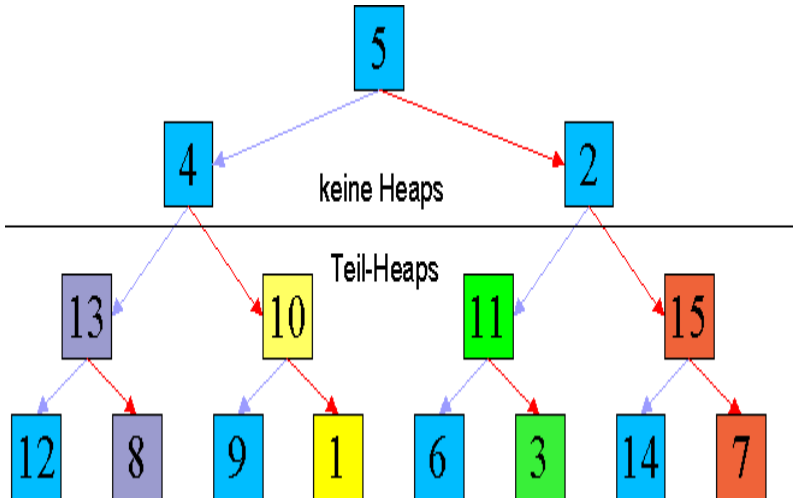
Heapsort

```
void Heap(CDatenSatz[] Feld) {
  if(Feld.length<=1) return;
  int Ende = Feld.length - 1;
  for(int Knoten = Ende/2; Knoten >= 0;
          Knoten--)
    Senke(Feld, Knoten, Ende);
  for(int Knoten=Ende; Knoten>=1; Knoten--){
    CDatenSatz swap = Feld[Knoten];
    Feld[Knoten] = Feld[0];
    Feld[0] = swap;
    Senke(Feld, 0, Knoten - 1);
} }
```



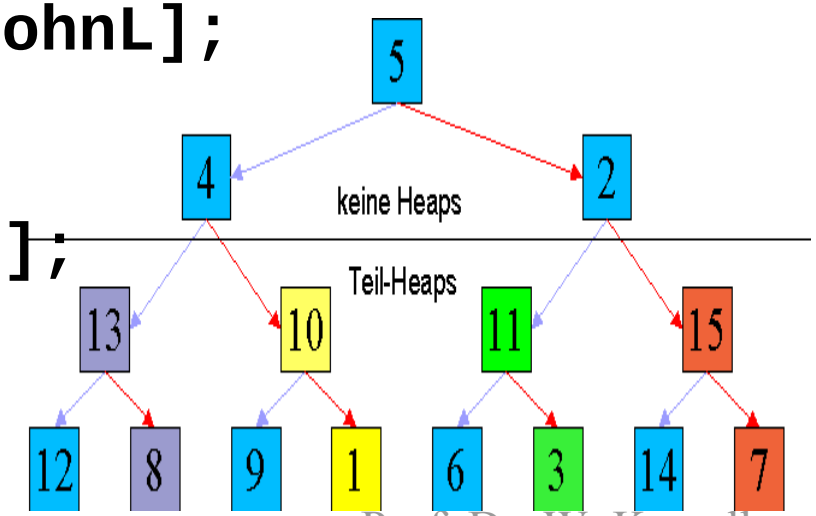
Heapsort

```
void SenkeUp(CDatenSatz[] Feld,
             int Vater, int Ende) {
  CDatenSatz swap = Feld[Vater];
  int Vater0 = Vater;
  int Sohn = Vater;
  int key = swap.key;
  int SohnL, SohnR;
```



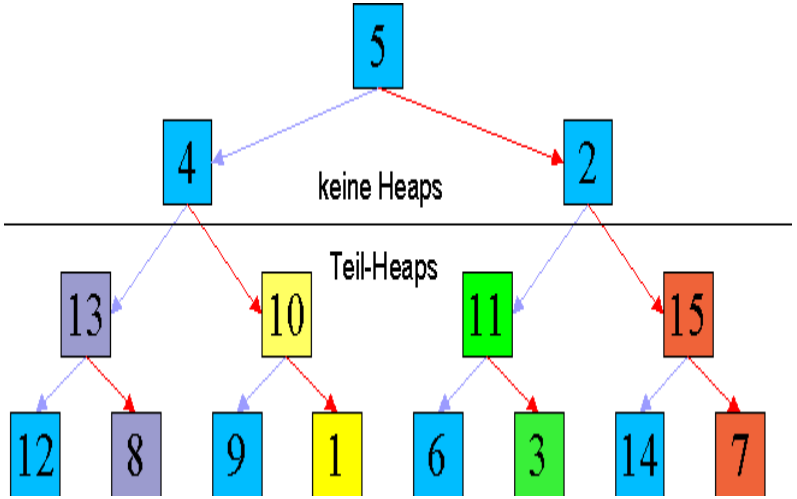
Heapsort

```
while (true) {
   SohnL = (Sohn << 1) + 1;
   if (SohnL > Ende) break;
   SohnR = SohnL + 1;
   if (SohnR <= Ende) {
      if (Feld[SohnL].key < Feld[SohnR].key) {
         Feld[Sohn]=Feld[SohnR];
         Sohn = SohnR;
      } else {
         Feld[Sohn] = Feld[SohnL];
         Sohn = SohnL;
   } } else {
      Feld[Sohn]=Feld[SohnL];
      Sohn = SohnL;
} }
```



Heapsort

```
  while (true) {
    if (Sohn == 0) {
      Feld[0]= swap;
      return;
    }
    Vater = (Sohn-1)/2;
    if (Feld[Vater].key < swap.key) {
      Feld[Sohn]=Feld[Vater];
      Sohn = Vater;
    } else {
      Feld[Sohn]=swap;
      return;
} } }
```



Heapsort

```
Void HeapUp(CDatenSatz[] Feld) {
  if(Feld.length<=1) return;
  int Ende = Feld.length - 1;
  for(int Knoten=Ende/2; Knoten>=0; Knoten--)
    Senke(Feld, Knoten, Ende);
    for(int Knoten=Ende; Knoten>=1; Knoten--) {
      CDatenSatz swap = Feld[Knoten];
      Feld[Knoten] = Feld[0];
      Feld[0] = swap;
      SenkeUp(Feld, 0, Knoten - 1);
}   }
```