Informatik II

1. Algorithmen und Datenstukturen

1.1. Terminologie

Algorithmus: Wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (input/Probleminstanz) Ausgabedaten (output) berechnet.

Datenstrukturen: Eine Datenstruktur organisiert Daten so in einem Computer, dass man sie (in den darauf operierenden Algorithmen) effizient nutzen kann.

Effizienz: Die Effizienz eines Algorithmus ist seine Sparsamkeit bezüglich der Ressourcen, Zeit und Speicherplatz, die er zur Lösung eines festgelegten Problems beansprucht.

1.2. Effizienz von Algorithmen

Asymptotische Laufzeiten

- **Obere Schranke**: $\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$
- **Untere Schranke**: $\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$
- **Scharfe Schranke**: $\Theta(g) := \Omega(g) \cap \mathcal{O}(g)$
  $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists c > 0,\ \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\}$

Theorem
Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ zwei Funktionen. Dann gilt.

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subsetneq \mathcal{O}(g)$
2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$ (C konstant) $\Rightarrow f \in \Theta(g)$
3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} \longrightarrow \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f)$

Beispiel aufsteigende Laufzeiten:

2^{16}, $\log(n^4)$, $\log^8(n)$, $\sqrt{n}$, $n\log n$, $\binom{n}{3}$, n^5+n, $\frac{2^n}{n^2}$, $n!$, n^n

1.3. Analyse mit Rekurenz und Teleskopie

```
void g(int n) {
    if (n>1) {
        g(n/2);
        g(n/2);
    }
    else {
        f();
    }
}
```

Rekurrenz ($n = 2^i$)

$$T(n) = \begin{cases} 2T(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

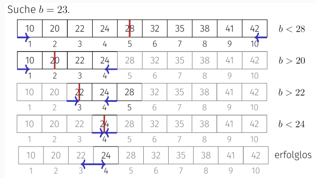
Teleskopieren

$T(n) = 2 \cdot T(n/2) = 2 \cdot (2 \cdot T(n/4))$
$= 2^i \cdot T(n/2^i) = n \cdot T(n/n) \in \Theta(n)$

2. Suchen

2.1. Divide and Conquer

Gegeben:	Sortiertes Array A mit n Elementen und einen Schlüssel b
Gesucht:	Index k mit $A[k] = b$
Lösung:	Zeiger und Halbierung des Arrays



2.2. Binärer Suchalgorithmus: BSearch(A,l,r,b)

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b. Bereichsgrenzen $1 \leq l, r \leq n$ mit $l \leq r$ oder $l = r + 1$.
Output: Index $m \in [l, \ldots, r+1]$, so dass $A[i] \leq b$ für alle $l \leq i < m$ und $A[i] \geq b$ für alle $m < i \leq r$.

$m \leftarrow \lfloor (l+r)/2 \rfloor$
if $l > r$ **then** // erfolglose Suche
  return l
else if $b = A[m]$ **then** // gefunden
  return m
else if $b < A[m]$ **then** // Element liegt links
  return BSearch$(A, l, m-1, b)$
else // $b > A[m]$: Element liegt rechts
  return BSearch$(A, m+1, r, b)$

3. Sortieren

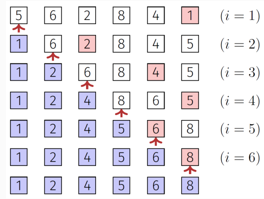
3.1. Laufzeiten von Sortier-Algorithmen

Algorithmus	Vergleiche		Vertauschungen	
	average	worst	average	worst
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Auswahl	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$
Einfügen	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n\log n)$	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)^\star$
Mergesort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$

3.2. Bubble-Sort

```
void bubbleSort(std::vector<T> vec) {
    n = vec.size();
    for (unsigned int i = 0; i < n; ++i) {
        for (unsigned int j = 0; j < n - i - 1; ++j) {
            if (vec[j] > vec[j+1]) {
                std::swap(arr[j], arr[j+1]);
            }
}}}
```

3.3. Sortieren durch Auswahl



Auswahl des kleinsten Elementes durch Suche im unsortierten Teil A[i:n] des Arrays.

Tausche kleinstes Element an das erste Element des unsortierten Teiles.

Unsortierter Teil wird ein Element kleiner ($i \to i+1$). Wiederhole bis alles sortiert.

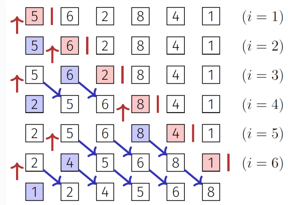
Selection Sort

Input: Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A
for $i \leftarrow 1$ **to** $n-1$ **do**
  $p \leftarrow i$
  for $j \leftarrow i+1$ **to** n **do**
    if $A[j] < A[p]$ **then**
      $p \leftarrow j$;
  swap$(A[i], A[p])$

Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$
Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$

3.4. Sortieren durch Einfügen



Iteratives Vorgehen: $i = 1 \to n$

Einfügeposition für Element i bestimmen.

Element i einfügen, ggfs. Verschiebung nötig.

Selection Sort

Input: Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A
for $i \leftarrow 2$ **to** n **do**
  $x \leftarrow A[i]$
  $p \leftarrow \texttt{BinarySearch}(A, 1, i-1, x)$; // Kleinstes $p \in [1, i]$ mit $A[p] \geq x$
  for $j \leftarrow i-1$ **downto** p **do**
    $A[j+1] \leftarrow A[j]$
  $A[p] \leftarrow x$

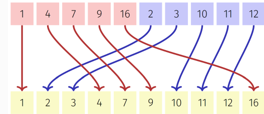
Analyse

Nachteil: Im schlechtesten Fall viele Elementverschiebungen. / Vorteil: Der Suchbereich (Einfügebereich) ist bereits sortiert → binäre Suche möglich.
Anzahl Vergleiche im schlechtesten Fall: $\Theta(n\log(n))$
Anzahl Vertauschungen im schlechtesten Fall: $\Theta(n^2)$

3.5. Mergesort

3.5.1. Merge

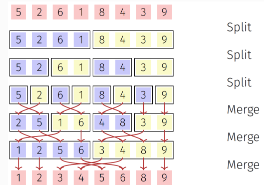
Minimum von A kann mit 2 Vergleichen ermittelt werden.



Merge(A,l,m,r)

Input: Array A der Länge n, Indizes $1 \leq l \leq m \leq r \leq n$. $A[l, \ldots, m]$, $A[m+1, \ldots, r]$ sortiert
Output: $A[l, \ldots, r]$ sortiert
$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m+1$; $k \leftarrow 1$
while $i \leq m$ and $j \leq r$ **do**
  if $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i+1$
  else $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
  $k \leftarrow k+1$;
while $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $k \leftarrow k+1$
while $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j+1$; $k \leftarrow k+1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k-l+1]$

3.5.2. Mergesort



Mergesort(A,l,r) → rekursive Variante

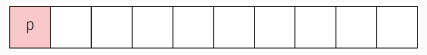
Input: Array A der Länge n. $1 \leq l \leq r \leq n$
Output: $A[l, \ldots, r]$ sortiert.
if $l < r$ **then**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$ // Mittlere Position
  $\texttt{Mergesort}(A, l, m)$ // Sortiere vordere Hälfte
  $\texttt{Mergesort}(A, m+1, r)$ // Sortiere hintere Hälfte
  $\texttt{Merge}(A, l, m, r)$ // Verschmelzen der Teilfolgen

Analyse: Laufzeit $\Theta(n\log(n))$; zusätzlicher Speicherbedarf: $\Theta(n)$

3.6. Quicksort

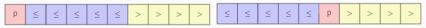
Pivotieren

1. Wähle ein beliebiges ELement als Pivot



2. Teile A in zwei Teile

3. Quicksort: Rekursion auf L und R



Wahl des Pivot

Maximum/Minimum (worst case) in $\mathcal{O}(n^2)$.
Best case (Pivot in der Mitte) $\omega(n)$.

Partition(A,l,r,p)

Input: Array A, welches den Pivot p in $A[l, \dots, r]$ mindestens einmal enthält.
Output: Array A partitioniert in $A[l, \dots, r]$ um p. Rückgabe der Position von p.
while $l \leq r$ **do**
  while $A[l] < p$ **do**
    $l \leftarrow l + 1$
  while $A[r] > p$ **do**
    $r \leftarrow r - 1$
  swap($A[l]$, $A[r]$)
  if $A[l] = A[r]$ **then**
    $l \leftarrow l + 1$
return l-1

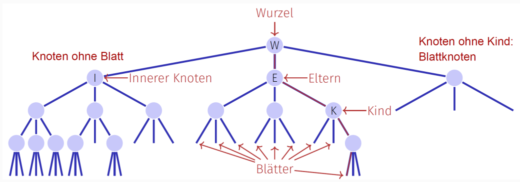
Quicksort(A,l,r)

Input: Array A der Länge n. $1 \leq l \leq r \leq n$.
Output: Array A, sortiert in $A[l, \dots, r]$.
if $l < r$ **then**
  Wähle Pivot $p \in A[l, \dots, r]$
  $k \leftarrow \texttt{Partition}(A, l, r, p)$
  $\texttt{Quicksort}(A, l, k-1)$
  $\texttt{Quicksort}(A, k+1, r)$

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log(n))$ Vergleiche.
Im schlechtesten Fall: $\Theta(n^2)$

4. Natürliche Suchbäume

4.1. Nomenklatur



4.2. Binäre Suchbäume

Binärer Baum (nur zwei Nachfolgerknoten) mit Eigenschaften:

- Jeder Knoten v speichert einen Schlüssel
- Schlüssel im linken Teilbaum v.left kleiner als v.key
- Schlüssel im rechten Teilbaum v.right grösser als v.key

4.2.1. Höhe eines Baumes

$$h(r) = \begin{cases} 0 & \text{falls } r = \text{ null} \\ 1 + \max\{h(r \cdot \text{left}), h(r \cdot \text{ right })\} & \text{sonst.} \end{cases}$$

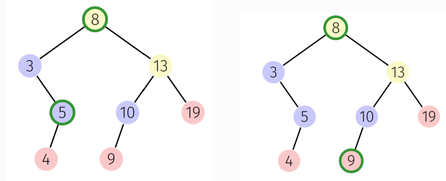
Die Laufzeit der Suche ist somit im schlechtesten Fall $\mathcal{O}(h(T))$.

4.2.2. Operationen

Knoten entfernen

Mögliche Situationen: Knoten hat keine Kinder, Knoten hat ein Kind oder Knoten v hat zwei Kinder. Im letzten Fall: Der kleinste Schlüssel im rechten Teilbaum $v.right$ ist der symmetrische Nachfolger von $v \rightarrow$ ersetze v durch seinen symmetrischen Nachfolger.
Auch möglich: ersetze v durch seinen symmetrischen Vorgänger
Implementation: der Teufel steckt im Detail!

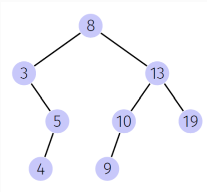
Grafik zu dem symmetrischen Vorgänger (links) oder Nachfolger (rechts)



4.3. Traversierungsarten

- **Hauptreihenfolge** (preorder):
  v, dann $T_{left}(v)$, dann $T_{right}(v)$.
- **Nebenreihenfolge** (postorder):
  $T_{left}(v)$, dann $T_{right}(v)$, dann v.
- **Symmetrische Reihenfolge** (inorder):
  $T_{left}(v)$, dann v, dann $T_{right}(v)$.

Beispiel



- **Hauptreihenfolge** (preorder):
  8, 3, 5, 4, 13, 10, 9, 19
- **Nebenreihenfolge** (postorder):
  4, 5, 3, 9, 10, 19, 13, 8
- **Symmetrische Reihenfolge** (inorder):
  3, 4, 5, 8, 9, 10, 13, 19

4.4. C++ - Implementation

4.4.1. Tree

```cpp
class BST {
  Node* root;
public:
  BST();
  bool contains(int key) const;
  bool insert(int key);
  bool remove(int key);
  void print_preorder(std::ostream& out) const;
  void clear();
  ~BST();
};

// Constructor, creates an empty binary search tree
BST::BST(): root(nullptr) {}

// Returns true iff the BST contains the given key.
bool BST::contains(int key) const {
  return root == nullptr ? false : root->contains(key);
}

// Returns true iff the given key was inserted into the BST.
bool BST::insert(int key) {
  if (root == nullptr) {
    root = new Node(key);
    return true;
  } else {
    return root->insert(key);
  }
}

// Returns true iff the given key was removed from the BST.
bool BST::remove(int key) {
  if (root == nullptr) {
    return false;
  } else if (root->key == key) {
    root = Node::symmetric_successor(root);
    return true;
  } else {
    return root->remove(key);
  }
}

// Prints the preorder-traversal of the BST to out.
void BST::print_preorder(std::ostream& out) const {
  if (root != nullptr) {
    root->print_preorder(out);
    out << '\n';
  }
}

// Clears the BST, i.e. resets it to empty.
void BST::clear() {
  delete root;
  root = nullptr;
}

// Deconstructor
BST::~BST() {
  clear();
}
```

4.4.2. Node

```cpp
struct Node {
  int key;     // Key, i.e. value stored in this node
  Node* left;  // Left child, i.e. root of left subtree
  Node* right; // Right child, i.e. root of right subtree

  Node(int key, Node* left, Node* right);

  bool contains(int search_key) const;
  bool insert(int new_key);
  bool remove(int remove_key);
  void print_preorder(std::ostream& out) const;
  static Node* symmetric_successor(Node* root);
  ~Node();
};

// Node constructor
Node::Node(int key, Node* left, Node* right):
       key(key), left(left), right(right) {}

// POST: Returns true iff the tree contains search_key.
bool Node::contains(int search_key) const {
  const Node* curr = this;

  while (curr != nullptr) {
    if (curr->key == search_key) {
      return true;
    } else if (search_key < curr->key) {
      curr = curr->left;
    } else {
      assert(search_key > curr->key);
      curr = curr->right;
    }
  }
  return false;
}

// POST: Inserts new_key into the tree, if not already
//  present. Returns true iff new_key was inserted.
//  Maintains the binary search-tree invariant.
bool Node::insert(int new_key) {
  Node* curr = this;

  while (curr->key != new_key) {
    if (new_key < curr->key) {
      if (curr->left == nullptr) {
        curr->left = new Node(new_key);
        return true;
      } else {
        curr = curr->left;
      }
    } else {
      assert(new_key > curr->key);
      if (curr->right == nullptr) {
        curr->right = new Node(new_key);
        return true;
      } else {
        curr = curr->right;
      }
    }
  }
  return false;
}

// PRE:  this->key != remove_key, i.e. remove_key may be
//  contained in this node's subtrees, but not in this node
//  directly.
// POST: If this tree contained remove_key, then the corre-
//  sponding node was removed, replaced by its symmetric
//  successor, and true is returned. Otherwise, nothing was
//  changed, and false is returned.
//  Maintains the binary search-tree invariant.
bool Node::remove(int remove_key) {
  assert(this->key != remove_key);

  // curr descends down the tree, while we look for the a
  // node with remove_key.
  // curr's key itself is always different from remove_key.
  Node* curr = this;
  Node* remove_node = nullptr;

  while (curr != nullptr && remove_node == nullptr) {
    if (
      curr->left != nullptr&& curr->left->key == remove_key){
      // curr's left child holds remove_key, and is replaced
      // with its symmetric successor.
      remove_node = curr->left;
      curr->left = symmetric_successor(curr->left);
    } else if (
      curr->right != nullptr &&
      curr->right->key == remove_key) {
      // Analogously, for curr's right child
      remove_node = curr->right;
      curr->right = symmetric_successor(curr->right);
    } else if (remove_key < curr->key) {
      // Descend left
      curr = curr->left;
    } else {
      // Descend right
      assert(remove_key > curr->key);
      curr = curr->right;
    }
  }

  if (remove_node != nullptr) {
    // To prevent memory leaks, the removed node is freed
    remove_node->left = nullptr;
    remove_node->right = nullptr;
    delete remove_node;
    return true;
  } else {
    return false;
  }
}

// POST: Prints the tree's keys in preorder traversal to out.
void Node::print_preorder(std::ostream& out) const {
  out << key << ' ';
  if (left != nullptr) left->print_preorder(out);
  if (right != nullptr) right->print_preorder(out);
}

// PRE:  root != nullptr
// POST: Returns the symmetric successor of the given root
//  node. If the symmetric successor is not a direct child of
//  the given root, then the symmetric successor is also
//  removed from its context, i.e. from the subtree of root
//  where the symmetric successor was found.
Node* Node::symmetric_successor(Node* root) {
```

```
  assert(root != nullptr);

  // If there's at most one child node, it must be the
  // symmetric successor, and we're done right away.
  if (root->left == nullptr) return root->right;
  if (root->right == nullptr) return root->left;

  // Otherwise, the symmetric successor must be the left-most
  // element of root's right subtree. We use curr to descend
  // down the tree, and parent will have parent->left == curr
  // (if parent != nullptr). parent needed for removing the
  // eventually found symmetric successor from its context.
  Node* curr = root->right;
  Node* parent = nullptr;

  // Descend leftwards. After the loop, curr is the symmetric
  // successor.
  while (curr->left != nullptr) {
    parent = curr;
    curr = curr->left;
  }

  // Remove the symmetric successor from its context, if
  // necessary.
  if (parent != nullptr) {
    parent->left = curr->right;
    curr->right = root->right;
  }

  curr->left = root->left;
  return curr;
}

// Deconstructor. All transitive children are recursively
// deleted.
Node::~Node() {
  delete left;
  delete right;
}
```

5. Heaps

5.1. [Max-]Heap

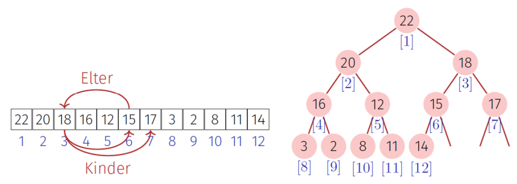
Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren.

Binärer Baum mit folgenden Eigenschaften

1. Vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. **Heap-Bedingung**:
   Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Elternknotens

Baum $\rightarrow$ Array:

1. Kinder $(i) = \{2i, 2i+1\}$
2. Elter $(i) = \lfloor i/2 \rfloor$



Höhe eines Heaps: $H(n) = \lceil \log_2(n+1) \rceil$

5.2. Heap bauen

- Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap. $\rightarrow$ Induktion von unten!
- Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \cdot \log(n))$.
- Versickerpfade sind aber im Mittel viel kürzer: $\mathcal{O}(n)$

5.3. Einfügen

- Füge neues Element an erste freie Stelle ein.
- Stelle Heap Eigenschaft wieder her: **Sukzessives Aufsteigen**
- Anzahl Operationen im worst case: $\mathcal{O}(\log(n))$

Aufsteigen(A,m)

Input: Array A mit mindestens m Elementen und Max-Heap-Struktur auf $A[1,\dots,m-1]$
Output: Array A mit Max-Heap-Struktur auf $A[1,\dots,m]$.
$v \leftarrow A[m]$ // Wert
$c \leftarrow m$ // derzeitiger Knoten (child)
$p \leftarrow \lfloor c/2 \rfloor$ // Elternknoten (parent)
while $c > 1$ and $v > A[p]$ **do**
  $A[c] \leftarrow A[p]$ // Wert Elternknoten $\rightarrow$ derzeitiger Knoten
  $c \leftarrow p$ // Elternknoten $\rightarrow$ derzeitiger Knoten
  $p \leftarrow \lfloor c/2 \rfloor$
$A[c] \leftarrow v$ // Wert $\rightarrow$ Wurzel des (Teil-)Baumes

5.4. Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: **Sukzessives Absinken** (in Richtung des grösseren Kindes / "Max. aufsteigen lassen")
- Anzahl Operationen im worst case: $\mathcal{O}(\log(n))$

Versickern(A,i,m)

Input: Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output: Array A mit Heapstruktur für i mit letztem Element m.
while $2i \le m$ **do**
  $j \leftarrow 2i$; // j linkes Kind
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j+1$; // j rechtes Kind mit grösserem Schlüssel
  if $A[i] < A[j]$ **then**
    swap$(A[i], A[j])$
    $i \leftarrow j$; // weiter versickern
  else
    $i \leftarrow m$; // versickern beendet

5.5. Heap sortieren

Worst case: $\mathcal{O}(n \cdot \log(n))$

HeapSort(A,n)

Input: Array A der Länge n.
Output: A sortiert.
// Heap aufbauen
for $i \leftarrow n/2$ **downto** 1 **do**
  Versickere(A, i, n)
// Nun ist A ein Heap
for $i \leftarrow n$ **downto** 2 **do**
  Vertausche$(A[1], A[i])$
  Versickere$(A, 1, i-1)$
// Nun ist A sortiert.

6. AVL Bäume

Ziel: Verhinderung der Degenerierung $\rightarrow$ garantiere, dass ein Baum mit n Knoten stets eine Höhe von $\mathcal{O}(\log(n))$.

6.1. AVL Bedingung

6.1.1. Balance eines Knotens

Die Balance eines Knotens v ist definiert als die Höhendifferenz seiner beiden Teilbäume $T_l(v)$ und $T_r(v)$:

$$\mathrm{bal}(v) := h(T_r(v)) - h(T_l(v))$$

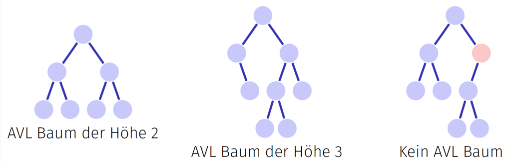
- **Augmentieren**: $v.size$ Feld mit **Balance**
- **Baumhöhe**: Ein AVL Baum ist asymptotisch nicht mehr als 44% höher als ein perfekt balancierter Baum ($\lceil \log_2 n + 1 \rceil$).

6.1.2. AVL Bedingung

AVL Bedingung: für *jeden* Knoten v eines Baumes gilt:

$$\mathrm{bal}(v) \in \{-1, 0, 1\}$$

Beispiele



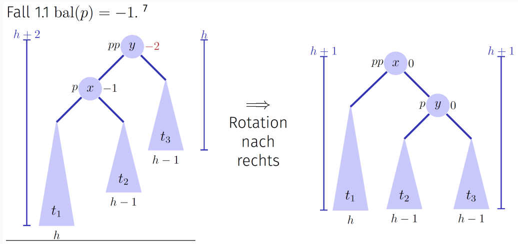
6.2. Einfügen

- Zuerst einfügen wie bei Suchbaum.
- Prüfe die Balance-Bedingung für alle Knoten aufsteigend von n zur Wurzel.
  $\mathrm{upin}(p)$: Aufsteigend von p die Balance (Augmentation) anpassen, wobei gilt $\mathrm{bal}(p) \in \{-1, 0, +1\}$.
- Problematischer Fall (rebalancieren!): p ist linker Sohn von pp, wobei $\mathrm{bal}(pp)$ bereits vor dem Einfügen -1 ist (danach -2).

6.3. Rebalancieren: Rotationen

Fall 1: Rotation nach rechts

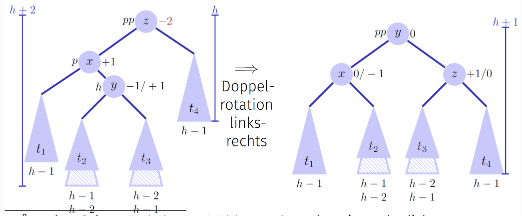
Fall 1.1 $\mathrm{bal}(p) = -1$. [7]



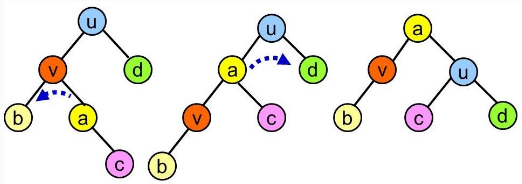
[7] p rechter Sohn $\Rightarrow \mathrm{bal}(pp) = \mathrm{bal}(p) = +1$, Linksrotation

Fall 2: Doppelrotation nach links-rechts

Fall 1.2 $\mathrm{bal}(p) = +1$. [8]



[8] p rechter Sohn $\Rightarrow \mathrm{bal}(pp) = +1$, $\mathrm{bal}(p) = -1$, Doppelrotation rechts links



6.4. Analyse

AVL-Bäume haben asymptotische Laufzeit von $\mathcal{O}(\log(n))$ (schlechtester Fall) für das Suchen, Einfügen und Löschen von Schlüsseln. Einfügen und Löschen ist verhältnismässig aufwändig und für kleine Probleme relativ langsam.

7. Hashing

7.1. Motivation und Idee

Key/Value-Paare effizient abspeichern und finden z.B. für Implementation eines Dicts oder einer Datenbank.

Wörterbuch $\longrightarrow$

```
fruits = {
  "banana": 2.95, "kiwi": 0.70,
  "pear": 4.20, "apple": 3.95
}
```

Idee

Direkter Zugriff (Array)

Probleme

1. **Schlüssel müssen nichtnegative ganze Zahlen sein**
2. **Grosser Schlüsselbereich $\rightarrow$ grosses Array**

7.2. Pre-Hashing: Lösung des ersten Problems

Prehashing: Bilde Schlüssel ab auf positive Ganzzahlen mit einer Funktion $ph : \mathcal{K} \to \mathbb{N}$

Pre-Hashing: Beispiel String
Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ zu Schlüssel

$$\mathrm{ph}(s) = \left(\sum_{i=0}^{l_s-1} s_{l_s-i} \cdot b^i\right) \bmod 2^w$$

b so, dass verschiedene Namen möglichst verschiedene Schlüssel erhalten.
w Wortgrösse des Systems (z.B. 32 oder 64).

```
#include <string>
unsigned prehash(std::string s) {
    unsigned b = B;
    unsigned h = 0;

    for (unsigned i = 0; i < s.size(); ++i){
        h = h * b + s[i];
    }
    return h;
}
```

7.3. Hashing: Lösung des zweiten Problems

Reduziere des Schlüsseluniversum: Abbildung (Hash-Funktion) $h : \mathcal{K} \to \{0, \dots, m-1\} (m \approx n =$ Anzahl Einträge in der Tabelle)

7.3.1. Nomenklatur

Hashfunktion h: Abbildung aus der Menge der Schlüssel $\mathcal{K}$ auf die Indexmenge $\{0, 1, \dots, m-1\}$ eines Arrays (**Hashtabelle**)
Meist $|\mathcal{K}| \gg m$, Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (**Kollision**). Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

7.3.2. Gebräuchliche Hashfunktion: Divisionsmethode

$$h(k) = k \bmod m$$

Ideal: m Primzahl, nicht zu nahe bei Potenzen von 2 oder 10
Aber oft: $m = 2^k - 1 (k \in \mathbb{N})$

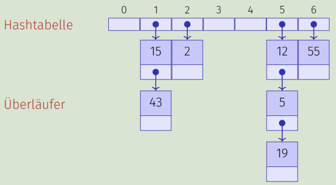
7.4. Konzept 1: Hashing mit Verkettung

Direkte Verkettung der Überläufer.
$\to$ Resultiert im worst case in $\Theta(n^2)$ pro Operation

Beispiel

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43
Direkte Verkettung der Überläufer



Einfaches gleichmässiges Hashing
Starke Annahmen: Jeder beliebige Schlüssel wird mit gleicher Wahrscheinlichkeit (**Uniformität**) und unabhängig von den anderen Schlüsseln (**Unabhängigkeit**) auf einen der m verfügbaren Slots abgebildet.

Unter dieser Annahme ergibt sich die **erwartete Länge**:
$\mathbb{E}($ Länge Kette $j) = \frac{n}{m} = \alpha$, α heisst der **Belegungsfaktor** oder **Füllgrad**.

Daraus ergibt sich (bei einfachem gleichmässigem Hashing) eine **erwartete Laufzeit (amortisiert)** von $\mathcal{O}(1)$ für Suchen, Einfügen, Löschen.

Vor- und Nachteile der Verkettung
- Belegungsfaktoren $\alpha > 1$ möglich; Entfernen von Schlüsseln einfach
- Speicherverbrauch der Verkettung

7.5. Konzept 2: Hashing mit offener Addressierung

- Speichere die Überläufer direkt in der Hashtabelle mit einer **Sondierungsfunktion** $s(k, j)$
- Tabellenposition des Schlüssels entlang der **Sondierungsfolge** $S(k)$

Technisches Detail zu **delete(k)**: Suche k in der Tabelle gemäss $S(k)$. Ersetze k durch den **speziellen Schlüssel removed**.

7.5.1. Lineares Sondieren

$$s(k, j) = h(k) + j \Rightarrow$$
$$S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \bmod m$$

Problem $\to$ **Primäre Häufung**:
Ähnliche Hashaddressen haben ähnliche Sondierungsfolgen $\to$ lange zusammenhängende belegte Bereiche.

Beispiel

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
5	15	2	19		12	55

7.5.2. Quadratisches Sondieren

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \bmod m$$

Problem $\to$ **Sekundäre Häufung**:
Synonyme k und k' (mit $h(k) = h(k')$) durchlaufen dieselbe Sondierungsfolge.

Beispiel

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
19	15	2		5	12	55

7.5.3. Double Hashing

Verwendung von zwei Hashfunktionen $h(k)$ und $h'(k)$ $\to$ Vermeidung primärer und sekundären Häufungen.

$$s(k, j) = h(k) + j \cdot h'(k)$$
$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k)b, \dots, h(k) + (m-1)h'(k)) \quad \bmod m$$

Gleichmässiges Hashing
Starke Annahme: Die Sondierungssequenz $S(k)$ eines Schlüssels k ist mit gleicher Wahrscheinlichkeit eine der $m!$ vielen Permutationssequenzen von $\{0, 1, \dots, m-1\}$. $\to$ Füllgrad $\alpha = \frac{n}{m} < 1$, so hat die nächste Operation erwartete Laufzeitkosten von $\leq \frac{1}{1-\alpha}$

Beispiel

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod 7, h'(k) = 1 + k \bmod 5.$

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
5	15	2	19		12	55

7.5.4. Beispiele

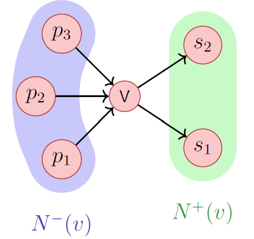
- $h'(k) = \lceil \ln(k+1) \rceil \bmod q$: This function is not suitable as a second hash function, because for the key $k = 0$ we have $h'(0) = \lceil \ln(1) \rceil = 0$.
- $s(j, k) = k^j \bmod p$: This function is not suitable as a probing function, because for the keys $k = 0$ and $k = 1$, the function $s(j, k)$ has constant value of 0 and 1.
- $s(j, k) = ((k \cdot j) \bmod q) + 1$: This function is also not suitable as a probing function because its value is constant 1 if the key k is a multiple of q.
  Moreover, for all other keys, the image of $s(j, k)$ is $\{1, \dots, q\}$, i.e., $p - q$ addresses of the hash table cannot be reached.

8. Graphen

8.1. Terminologie

Ein Graph $G = (V, E)$ besteht aus der Menge von Kanten $V = \{v_1, \dots, v_1 n\}$ und der Menge von Kanten E.

Gerichteter Graph: $E \subseteq V \times V = \{(u, v) : u \in V, v \in V\}$
- $w \in V$ heisst **adjazent** zu $v \in V$, falls $(v, w) \in E$
- Vorgänger eines Knotens v: $N^-(v) := \{u \in V | (u, v) \in E\}$
- Nachfolger eines Knotens v: $N^+(v) := \{u \in V | (v, u) \in E\}$
- Eingangsgrad: $\deg^-(v) := |N^-(v)|$
- Ausgangsgrad: $\deg^+(v) := |N^+(v)|$



Ungerichteter Graph: $E \subseteq \{(u, v) : v \in V, u \in V\}$
- $w \in V$ heisst **adjazent** zu $v \in V$, falls $\{v, w\} \in E$
- Nachbarschaft: $N(v) := \{w \in V | \{v, w\} \in E\}$
- Grad: $\deg(v) := |N(v)|$ (Schleifen zählen 2)

Vollständiger Graph: Ungerichteter Graph mit $E = \{(u, v) : u \in V, v \in V, \quad c \neq v\}$

Bipartiter Graph: Graph, bei dem V so in disjunkte U und W aufgeteilt werden kann, dass alle $e \in E$ einen Knoten in U und einen in W haben

Handschlag-Lemma:
- $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, falls G gerichtet
- $\sum_{v \in V} \deg(v) = 2|E|$, falls G ungerichtet

Wege:
- **Weg / Path**: Sequenz von Knoten $p = \langle v_1, v_2, \dots, v_k \rangle$ so dass für jedes $i \in \{1 \dots k\}$ eine Kante von v_i nach v_{i+1} existiert
- **Pfad / einfacher Pfad / simple path**: Weg der keinen Knoten mehrfach verwendet
- **Länge des Weges**: Anzahl enthaltene Kanten k
- **Gewicht des Weges** (in gewichteten Graphen): $\sum_{i=1}^k c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^k c(\{v_i, v_{i+1}\})$)

Zusammenhang:
- Ungerichteter Graph heisst **zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **stark zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **schwach zusammenhängend**, wenn der entsprechende ungerichtete Graph zusammenhängend ist.

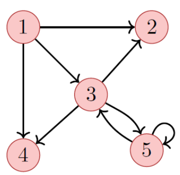
Zyklen:
- **Zyklus**: Weg (und nicht einfacher Pfad!) $\langle v_1, \dots, v_{k+1} \rangle$ mit $v_1 = v_{k+1}$
- **Einfacher Zyklus**: Zyklus, aber Knoten kommen nicht mehrfach vor (ausser s und t)
- **Kreis**: Zyklus mit paarweise verschiedenen $v_1, \dots, v_k$, welcher keine Kante mehrfach verwendet
- **Kreisfrei (azyklisch)**: Graph ohne jegliche Kreise.

Beobachtungen
- Allgemein: $0 \leq |E| \in \mathcal{O}\left(|V|^2\right)$
- Zusammenhängender Graph: $|E| \in \Omega(|V|)$
- Vollständiger Graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (ungerichtet)
- Maximal $|E| = |V|^2$(gerichtet)
- Maximal $|E| = \frac{|V| \cdot (|V|+1)}{2}$ (ungerichtet)

8.2. Repräsentation von Graphen

8.2.1. Adjazenzmatrix

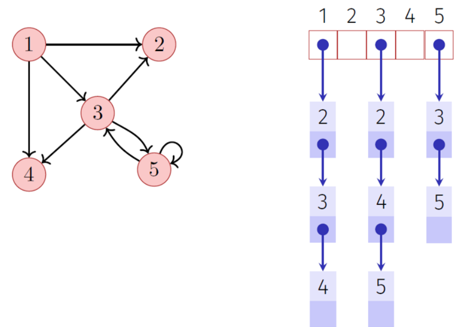
Graph $G = (V, E)$ mit Knotenmenge $v_1, \dots, v_n$ gespeichert als Adjazenzmatrix $A_G = (a_{ij})_{1 \leq i, j \leq n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j. Speicherbedarf $\Theta\left(|V|^2\right)$. A_G ist symmetrisch, wenn G ungerichtet.



$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

8.2.2. Adjazenzliste

Viele Graphen $G = (V, E)$ mit Knotenmenge $v_1, \dots, v_n$ haben deutlich weniger als n^2 Kanten. Repräsentation mit Adjazenzliste: Array $A[1], \dots, A[n]$, A_i enthält verkettete Liste aller Knoten in $N^+(v_i)$. Speicherbedarf $\Theta(|V| + |E|)$.

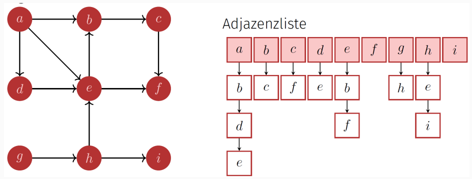


8.2.3. Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen	$\Theta(1)$	$\Theta(\deg^+ v)$

8.3. Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



Reihenfolge: $a, b, c, f, d, e, g, h, i$

Tiefensuche ab Knoten v: DFS-Visit(G,v)

Laufzeit (ohne Rekursion): $\Theta\left(\deg^+ v\right)$

Tiefensuche für alle Knoten: DFS-Visit(G)

Laufzeit: $\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$

DFS-Visit(G,v)

Input: Graph $G = (V, E)$

foreach $v \in V$ **do**
  $v.color \leftarrow$ white

foreach $v \in V$ **do**
  if $v.color =$ white **then**
    DFS-Visit(G,v)

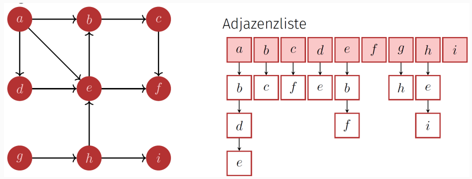
DFS-Visit(G)

Input: Graph $G = (V, E)$, Knoten v.

$v.color \leftarrow$ grey
foreach $w \in N^+(v)$ **do**
  if $w.color =$ white **then**
    DFS-Visit(G, w)
$v.color \leftarrow$ black

8.4. Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Reihenfolge: $a, b, d, e, c, f, g, h, i$

BFS-Visit(G,v)

Input: Graph $G = (V, E)$

Queue $Q \leftarrow \emptyset$
$v.color \leftarrow$ grey
enqueue(Q, v)
while $Q \neq \emptyset$ **do**
  $w \leftarrow$ dequeue(Q)
  foreach $c \in N^+(w)$ **do**
    if $c.color =$ white **then**
      $c.color \leftarrow$ grey
      enqueue(Q, c)
  $w.color \leftarrow$ black

BFS-Visit(G)

Input: Graph $G = (V, E)$

foreach $v \in V$ **do**
  $v.color \leftarrow$ white

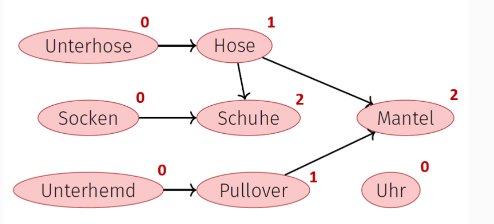
foreach $v \in V$ **do**
  if $v.color =$ white **then**
    BFS-Visit(G,v)

Extraplatz: $\mathcal{O}(|V|)$

Laufzeit: $\Theta(|V| + |E|)$

8.5. Topologische Sortierung

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er **kreisfrei** ist.



Augmentiere den Eingangsgrad. Abbarbeitung nur wenn Eingangsgrad 0 ist. Eingangsgrad verringern entspricht Knotenentfernen.

Topological-Sort(G)

Input: Graph $G = (V, E)$.
Output: Topologische Sortierung ord

Stack $S \leftarrow \emptyset$
foreach $v \in V$ **do** $A[v] \leftarrow 0$
foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Eingangsgrade berechnen
foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Merke Nodes mit Eingangsgrad 0
$i \leftarrow 1$
while $S \neq \emptyset$ **do**
  $v \leftarrow$ pop(S); ord$[v] \leftarrow i$; $i \leftarrow i + 1$ // Wähle Knoten mit Eingangsgrad 0
  foreach $(v, w) \in E$ **do** // Verringere Eingangsgrad der Nachfolger
    $A[w] \leftarrow A[w] - 1$
    if $A[w] = 0$ **then** push(S, w)

if $i = |V| + 1$ **then return** ord **else return** "Cycle Detected"

Analyse

- Sei $G = (V, E)$ ein gerichteter, kreisfreier Graph. Der Algorithmus Topological-Sort berechnet in Zeit $\Theta(|V| + |E|)$ eine topologische Sortierung ord für G.
- Sei $G = (V, E)$ ein gerichteter, **nicht**-kreisfreier Graph. Der Algorithmus Topological-Sort terminiert in Zeit $\Theta(|V| + |E|)$ und detektiert den Zyklus.

8.6. Kürzeste Wege

Notation

$\delta(u, v) =$ Gewicht eines kürzesten Weges von u nach v

$$\delta(u, v) = \begin{cases} \infty & \text{kein Weg von } u \text{ nach } v \\ \min\{c(p) : u \xrightarrow{p} v\} & \text{sonst} \end{cases}$$

Beobachtungen

- Einfachster Fall: Kantengewicht 1 $\rightarrow$ Breitensuche
- Es gibt Situationen, in denen kein kürzester Weg existiert: negative Zyklen könnten auftreten.
- Es kann exponentiell viele Wege geben $\rightarrow$ alle Wege probieren ist ineffizient
- Ein kürzester Weg von s nach v (ohne weitere Einschränkungen) kann nicht länger sein als ein kürzester Weg von s nach v, der u enthalten muss.
  $\delta(s, v) \leq \delta(s, u) + \delta(u, v)$
- **Optimale Substruktur**: Teilpfade von kürzesten Pfaden sind kürzeste Pfade (**Kürzester Pfad $\Rightarrow$ kürzeste Subpfäde**)
- Kürzeste Wege enthalten keine Zyklen

8.6.1. Allgemeiner Algorithmus (Relaxier-Algorithmus)

Gesucht: Kürzeste Wege von einem Startknoten s aus.

- Gewicht des kürzesten bisher gefundenen Pfades
  - Zu Beginn: $d_s[v] = \infty$ für alle Knoten $v \in V$
  - Ziel: $d_s[v] = \delta(s, v)$ für alle $v \in V$
- Vorgänger eines Knotens: u Beginn $\pi_s[v]$ undefiniert für jeden Knoten $v \in V$

Algorithmus

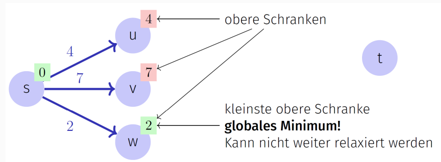
1. Initialisiere d_s und π_s : $d_s[v] = \infty$, $\pi_s[v] =$ null für alle $v \in V$
2. Setze $d_s[s] \leftarrow 0$
3. Wähle eine Kante $(u, v) \in E$:

   Relaxiere (u, v)**:**
   if $d_s[v] > d[u] + c(u, v)$ then
     $d_s[v] \leftarrow d_s[u] + c(u, v)$
     $\pi_s[v] \leftarrow u$

4. Wiederhole 3 bis nichts mehr relaxiert werden kann (bis $(d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E)$)

8.6.2. Dijkstra Algorithmus

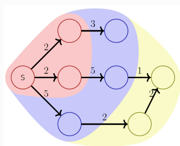
Beobachtung



Grundidee

Menge V aller Knoten wird unterteilt in

- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \cup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt wurden.



Betrachte alle Nachbarn der Menge M und füge den Knoten mit dem kürzesten Weg zu s der Menge M hinzu.

Dijkstra(G,s)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

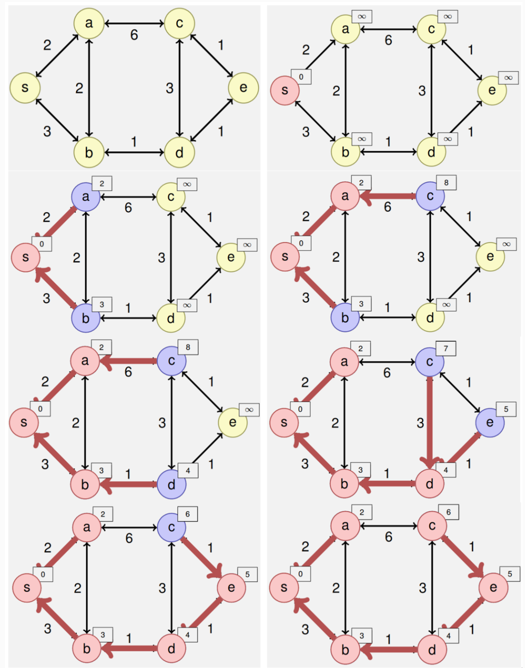
foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
while $R \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(R)
  foreach $v \in N^+(u)$ **do**
    if $d_s[u] + c(u, v) < d_s[v]$ **then**
      $d_s[v] \leftarrow d_s[u] + c(u, v)$
      $\pi_s[v] \leftarrow u$
      if $v \in R$ **then**
        DecreaseKey(R, v) // Update eines $d(v)$ im Heap zu R
      else
        $R \leftarrow R \cup \{v\}$ // Einfügen eines neuen $d(v)$ im Heap zu R

DecreaseKey (Aufsteigen im MinHeap), Position im Heap: Speichern am Knoten, Hashtabelle oder Lazy Deletion

Laufzeit

- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert oder DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- **Insgesamt**: $\mathcal{O}((|E| + |V|) \log |V|)$

Beispiel Dijkstra



8.7. Minimale Spannbäume

Problem

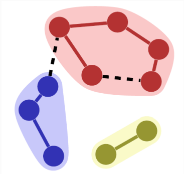
- Gegeben: Ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$
- Gesucht: Minimaler Spannbaum $T = (V, E')$: zusammenhängender, zyklenfreier Teilgraph $E' \subset E$, so dass $\sum_{e \in E'} c(e)$ minimal.

Greedy (gierige) Verfahren berechnen eine Lösung schrittweise, indem lokal beste Lösungen gewählt werden.

8.7.1. Union-Find Kruskal Algorithmus

Zur Implementation

Gegeben eine Menge von Mengen $i \equiv A_i \subset V$. Zur Identifikation von Schnitten und Kreisen: Zugehörigkeit der beiden Endpunkte einer Kante zu einer der Mengen.



Allgemeines Problem: Partition (Menge von Teilmengen) benötigt einen abstrakter Datentyp (**Union-Find**) mit folgenden Operationen:

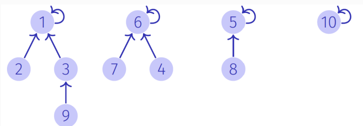
- Make-Set((i)): Hinzufügen einer neuen Menge i
  ($p[i] \leftarrow i; return i$)
- Find (e): Name i der Menge, welche e enthält
  ($(while(p[i] \neq 0) do\ i \leftarrow p[i]$; return i)
- Union(i, j): Vereingung der Mengen mit Namen i und j
  ($p[j] = i$, wobei i und j die Wurzeln (Namen) sind.)

Laufzeitoptimierungen:

a) Immer kleineren Baum an grösseren hängen
b) Bei Find Knoten immer an den Parent hängen

Implementation von Union-Find

Idee: Baum für jede Teilmenge in der Partition, z.B. 1, 2, 3, 9, 7, 6, 4, 5, 8, 10, wobei die Baumwurzeln → Namen (Stellvertreter) der Mengen ist.



Index	1	2	3	4	5	6	7	8	9	10
Parent	1	1	1	6	5	6	5	5	3	10

Algorithmus

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimaler Spannbaum mit Kanten A.

```
Sortiere Kanten nach Gewicht c(e_1) ≤ ... ≤ c(e_m)
A ← ∅
for k = 1 to |V| do
    MakeSet(k)
for k = 1 to m do
    (u, v) ← e_k
    if Find(u) ≠ Find(v) then
        Union(Find(u), Find(v))
        A ← A ∪ e_k
    else                                  // konzeptuell: R ← R ∪ e_k
return (V, A, c)
```

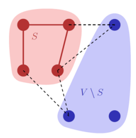
Laufzeit des Kruskal Alghorithmus

- Sortieren der Kanten: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$
- Initialisieren der Union-Find Datenstruktur $\Theta(|V|)$
- $|E| \times$ Union (Find (x), Find (y)) : $\mathcal{O}(|E| \log |V|)$
- **Insgesamt**: $\Theta(|E| \log |V|)$

8.7.2. Algorithmus von Jarnik, Prim, Dijkstra

Idee: Starte mit einem $v \in V$ und lasse von dort unter Verwendung der Auswahlregel einen Spannbaum wachsen:

```
A ← ∅
S ← {v_0}
for i ← 1 to |V| do
    Wähle billigste (u, v) mit u ∈ S, v ∉ S
    A ← A ∪ {(u, v)}
    S ← S ∪ {v}  // (Färbung)
```



Bemerkungen

- Man braucht keine Union-Find Datenstruktur (Färbung reicht aus)
- Vorgehensweise:
  a) Immer Knoten mit kleinstem Gewicht zur Menge S hinzufügen
  b) Wenn der Knoten noch nicht in S ist → MST ist zyklenfrei

Laufzeit insgesamt: $\mathcal{O}(|E| \log |V|)$

9. Dynamische Programmierung

9.1. Idee

- Aufteilen eines komplexen Problems in eine vernünftige Anzahl kleinerer Teilprobleme
- Die Lösung der Teilprobleme wird zur Lösung des komplexeren Problems verwendet
- Identische Teilprobleme werden nur einmal gerechnet

→ Wir tauschen Laufzeit gegen Speicherplatz

9.2. Dynamic Programming vs. Divide-And-Conquer

- **Optimale Substruktur**: In beiden Fällen ist das Ursprungsproblem (einfacher) lösbar, indem Lösungen von Teilproblemen herangezogen werden können.
- Bei Divide-And-Conquer Algorithmen sind **Teilprobleme unabhängig**; deren Lösungen werden im Algorithmus nur einmal benötigt.
  Beim DP sind Teilprobleme nicht unabhängig. Das Problem hat **überlappende Teilprobleme**, welche im Algorithmus mehrfach gebraucht werden.
- Identische Teilprobleme werden nur einmal gerechnet d.h. **keine zirkulären Abhängigkeiten zwischen Teilproblemen**

9.3. Memoization

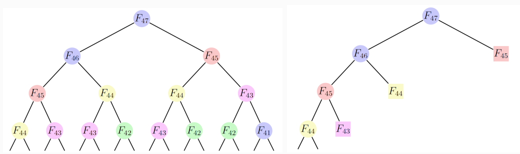
Memoization (sic) Abspeichern von Zwischenergebnissen.

- Bevor ein Teilproblem gelöst wird, wird Existenz eines entsprechenden Zwischenergebnis geprüft
- Existiert ein gespeichertes Zwischenergebnis bereits, so wird dieses verwendet.
- Andernfalls wird der Algorithmus ausgeführt und das Ergebnis wird entsprechend gespeichert

Beispiel Fibonacci

Input: $n \geq 0$
Output: n-te Fibonacci Zahl

```
if n ≤ 2 then
    f ← 1
else if ∃memo[n] then
    f ← memo[n]
else
    f ← FibonacciMemoization(n − 1) + FibonacciMemoization(n − 2)
    memo[n] ← f
return f
```



→ Genau hingesehen lösen wir das Problem bottom-up

9.4. Dynamic Programming: Beschreibung am Beispiel

Beispiel Fibonacci

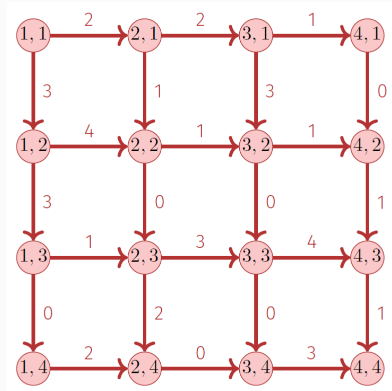
1. **Dimension der Tabelle? Bedeutung der Einträge?**
   Tabelle der Grösse $n \times 1$. n-ter Eintrag enthält n-te Fibonacci Zahl.
2. **Welche Einträge hängen nicht von anderen ab?**
   Werte F_1 und F_2 sind unabhängig einfach "berechenbar".
3. **Berechnungsreihenfolge?**
   F_i mit aufsteigenden i.
4. **Rekonstruktion einer Lösung?**
   F_n ist die n-te Fibonacci-Zahl.

9.5. Wie findet man den DP Algorithmus?

1. Genaue Formulierung der gesuchten Lösung
2. Definiere Teilprobleme (und bestimme deren Anzahl)
3. Raten / Aufzählen (und bestimme die Laufzeit für das Raten)
4. Rekursion: verbinde die Teilprobleme
5. Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme
6. Lösung des Problems:
   Laufzeit = Anz. Teilprobleme $\times$ $\frac{\text{Zeit}}{\text{Teilproblem}}$

9.6. Beispiel Kaninchen

Ein Kaninchen sitzt auf Platz (1,1) eines $n \times n$ Gitters. Es kann nur nach Osten oder nach Süden gehen. Auf jedem Wegstück liegt eine Anzahl Rüben. Wie viele Rüben sammelt das Kaninchen maximal ein?



Rekursion

Gesucht: $T_{0,0}$ = Maximale Anzahl Rüben von (0,0) nach (n, n) Sei $w_{(i,j)-(i',j')}$ Anzahl Rüben auf Kante von (i, j) nach (i', j') Rekursion (maximale Anzahl Rüben von (i, j) nach (n, n))

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i < n, j < n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i = n, j < n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

Teilabhängigkeitsgraph

- Richtung der Abhängigkeiten: Links oben nach rechts unten
- Richtung der Berechung: Rechts unten nach links oben

Bottom-Up Beschreibung am Beispiel

1. **Dimension der Tabelle? Bedeutung der Einträge?**
   Tabelle T der Grösse $n \times n$. Eintrag bei i, j enthält die maximale Anzahl Rüben von (i, j) nach (n, n).
2. **Welche Einträge hängen nicht von anderen ab?**
   Wert $T_{n,n}$ ist 0.
3. **Berechnungsreihenfolge?**
   $T_{i,j}$ mit $i = n \searrow 1$ und für jedes i: $j = n \searrow 1$, (oder umgekehrt: $j = n \searrow 1$ und für jedes j: $i = n \searrow 1$).
4. **Rekonstruktion einer Lösung?**
   $T_{1,1}$ enthält die maximale Anzahl Rüben

9.7. Die Editierdistanz / Levenshteinabstand

Aufgabenstellung

Gesucht: Günstigste zeichenweise Transformation $A_n \rightarrow B_m$ mit Kosten

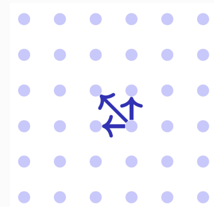
Operation	Levenshtein	LGT[24]	allgemein
c einfügen	1	1	ins(c)
c löschen	1	1	del(c)
Ersetzen $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

T	I	G	E	R		T	I	_	G	E	R	T→Z	+E	-R
Z	I	E	G	E		Z	I	E	G	E	_	Z→T	-E	+R

Wie findet man den DP Algorithmus?

1. Genaue Formulierung der gesuchten Lösung:
   $E(n, m)$ = minimale Anzahl Editieroperationen (ED Kosten) für $a_{1\ldots n} \rightarrow b_{1\ldots m}$
2. Definiere Teilprobleme (und bestimme deren Anzahl):
   Teilprobleme $E(i, j)$ = ED von $a_{1\ldots i}.b_{1\ldots j}$ (Anz. $n \cdot m$)
3. Raten / Aufzählen (und bestimme die Laufzeit für das Raten):
   $a_{1\ldots i} \rightarrow a_{1\ldots i-1}$(löschen) $a_{1\ldots i} \rightarrow a_{1\ldots i}b_j$ (einfügen) $a_{1\ldots i} \rightarrow a_{1\ldots i-1}b_j$ (ersetzen)
4. Rekursion: verbinde die Teilprobleme:
   $$E(i, j) = \min \begin{cases} \text{del}\ (a_i) + E(i-1, j) \\ \text{ins}\ (b_j) + E(i, j-1) \\ \text{repl}\ (a_i, b_j) + E(i-1, j-1) \end{cases}$$
5. Memoisieren / Tabellieren. Bestimme die Abhängigkeiten der Teilprobleme:

   Berechnung von links oben nach rechts unten. Zeilen- oder Spaltenweise.
6. Lösung des Problems: Lösung steht in $E(n, m)$

Bottom-Up Beschreibung

1. **Dimension der Tabelle? Bedeutung der Einträge?**
   Tabelle $E[0,\dots,m][0,\dots,n]$. $E[i,j]$: Minimaler Editierabstand der Zeichenketten $(a_1,\dots,a_i)$ und $(b_1,\dots,b_j)$
2. **Berechnung eines Eintrags**
   $E[0,i] \leftarrow i \;\forall 0 \le i \le m$, $E[j,0] \leftarrow i \;\forall 0 \le j \le n$. Berechnung von $E[i,j]$ sonst mit $E[i,j] = \min\{\text{del}(a_i)+E(i-1,j), \text{ins}(b_j)+E(i,j-1), \text{repl}(a_i,b_j)+E(i-1,j-1)\}$
3. **Berechnungsreihenfolge**
   Abhängigkeiten berücksichtigen: z.B. Zeilen aufsteigend und innerhalb von Zeilen Spalten aufsteigend.
4. **Rekonstruktion einer Lösung?**
   Beginne bei $j=m$, $i=n$. Falls $E[i,j] = \text{repl}(a_i,b_j)+E(i-1,j-1)$ gilt, gib $a_i \to b_j$ aus und fahre fort mit $(j,i) \leftarrow (j-1,i-1)$; sonst, falls $E[i,j] = \text{del}(a_i)+E(i-1,j)$ gib $\text{del}(a_i)$ aus fahre fort mit $j \leftarrow j-1$; sonst, falls $E[i,j] = \text{ins}(b_j)+E(i,j-1)$, gib $\text{ins}(b_j)$ aus und fahre fort mit $i \leftarrow i-1$. Terminiere für $i=0$ und $j=0$.

Analyse

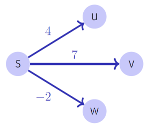
Anzahl Tabelleneinträge: $(m+1)\cdot(n+1)$
Laufzeit: $\mathcal{O}(m\cdot n)$

9.8. Kürzeste Wege: DP Ansatz (Bellman)

Induktion über Anzahl Kanten $d_s[i,v]$: Kürzeste Weglänge von s nach v über maximal i Kanten.

$$d_s[i,v] = \min\begin{cases} d_s[i-1,v] \\ \min_{(u,v)\in E}(d_s[i-1,u]+c(u,v)) \end{cases}$$
$$d_s[0,s] = 0, \underbrace{d_s[0,v] = \infty \forall v \ne s}_{\text{Zyklus}}$$

	s	$\cdots$	v	$\cdots$	w
0	0	∞	∞	∞	∞
1	0	∞	7	∞	-2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	0	$\cdots$	$\cdots$	$\cdots$	$\cdots$



Algorithmus: Iteriere über letzte Zeile bis die Relaxationsschritte keine Änderung mehr ergeben, maximal aber $n-1$ mal. Wenn dann noch Änderungen, dann gibt es keinen kürzesten Pfad.

Bellmann-Ford(G,s)

Input: Graph $G=(V,E,c)$, Startpunkt $s\in V$
Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ **to** $|V|$ **do**
  $f \leftarrow$ false
  foreach $(u,v)\in E$ **do**
    $f \leftarrow f \vee \text{Relax}(u,v)$
  if $f=$ false **then return** true
return false;

Analyse

Laufzeit: $\mathcal{O}(|V|\cdot|E|)$
Speicherplatz: $\mathcal{O}(|V|^2) \rightsquigarrow$ eigentlich sogar $\mathcal{O}(|V|)$, da nur immer die letzte Zeile abgespeichert werden muss.

10. Greedy-Algorithmen

10.1. Eigenschaften

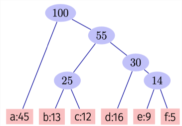
Ein rekursiv lösbares Optimierungsproblem kann mit einem **gierigen (greedy) Algorithmus** gelöst werden, wenn es die folgende Eigenschaften hat:

- Das Problem hat **optimale Substruktur**: die Lösung eines Problems ergibt sich durch Kombination optimaler Teillösungen.
- Es gilt die **greedy choice property**: Die Lösung eines Problems kann konstruiert werden, indem ein lokales Kriterium herangezogen wird, welches nicht von der Lösung der Teilprobleme abhängt.

10.2. Huffman-Code

Betrachten Präfixcodes: kein Codewort kann mit einem anderen Codewort beginnen.

Präfixcodes können im Vergleich mit allen Codes die optimale Datenkompression erreichen.



Huffman(C)

Input: Codewörter $c \in C$
Output: Wurzel eines optimalen Codebaums

$n \leftarrow |C|$
$Q \leftarrow C$
for $i=1$ **to** $n-1$ **do**
  Alloziere neuen Knoten z
  z.left $\leftarrow$ ExtractMin(Q)   // Extrahiere Wort mit minimaler Häufigkeit.
  z.right $\leftarrow$ ExtractMin(Q)
  z.freq $\leftarrow$ z.left.freq + z.right.freq
  Insert(Q,z)
return ExtractMin(Q)

Analyse: Heap bauen in $\mathcal{O}(n)$. Extract-Min in $\mathcal{O}(\log(n))$ für n Elemente. Somit Laufzeit $\mathcal{O}b(n\log(n))$.

11. Code Beispiele

11.1. Beispiel: Levenshtein-Algorithmus

```cpp
#include <string>
#include <vector>

unsigned Levenshtein(const std::string& x,
                     const std::string& y){
  // D[n,m] = distance between x and y
  // D[i,j] = distance between strings x[1..i] and y[1..j]
  unsigned n = x.size();
  unsigned m = y.size();
  std::vector<std::vector<unsigned>> D(
      n+1,std::vector<unsigned>(m+1,0)
    );
  for (unsigned j = 0; j <=m; ++j){
    D[0][j] = j;
  }
  for (unsigned i = 1; i <= n; ++i){
    D[i][0] = i;
    for (unsigned j = 1; j <=m; ++j){
      // D[i,j] = min{
      // D[i-1,j-1] + d(x[i],y[j]);
      // D[i-1,j] + 1
      // D[i,j-1] + 1 }
      D[i][j] = std::min( {
        D[i-1][j-1] + (x[i-1]!=y[j-1]),
        D[i][j-1] + 1,
        D[i-1][j] + 1
      });
    }
  }
  return D[n][m];
}
```

11.2. Beispiel: Längste gemeinsame Teilfolge

```python
import Data
import time

# compute longest ascending sequence for a point of the matrix
def LASR(A,L,y,x):
  if L[y][x] > 0:
    return L[y][x]
  maxLength = 0
  if x>0 and A[y][x] < A[y][x-1]:
    maxLength = max(maxLength, LASR(A,L,y,x-1))
  if y>0 and A[y][x] < A[y-1][x]:
    maxLength = max(maxLength, LASR(A,L,y-1,x))
  if y<len(A)-1 and A[y][x] < A[y+1][x]:
    maxLength = max(maxLength, LASR(A,L,y+1,x))
  if x<len(A[y])-1 and A[y][x] < A[y][x+1]:
    maxLength = max(maxLength, LASR(A,L,y,x+1))
  L[y][x] = maxLength + 1;
  return L[y][x]

# compute longest ascending sequence for each point of the matrix
def LAS(A):
  maxLength = 0
  L = [[0] * len(A[i]) for i in range(len(A))]
  for y in range(len(A)):
    for x in range(len(A[y])):
      L[y][x] = LASR(A,L,y,x)
      maxLength = max(maxLength, L[y][x])
  return maxLength,L

A = Data.get()

start = time.time()
(m,L) = LAS(A)
stop = time.time();

if len(A)<15 and len(A[0])<15:
  print("matrix a")
  Data.print_matrix(A)
  print("path lengths matrix")
  Data.print_matrix(L)

print("maximum length",m)
print("time:", stop-start, "s")
```

11.3. Beispiel: Union Find

```python
class Set:
  def __init__(self,n):
    self.a = [i for i in range(0,n)]
    self.g = [1] * n

  def union(self,i,j):
    i = self.find(i)
    j = self.find(j)
    if i == j:
      return False
    self.a[j] = i # j under i
    return True

  def find(self,i):
    while self.a[i] != i:
      i = self.a[i]
    return i

  def path_length(self,i):
    count = 1;
    while self.a[i] != i:
        i = self.a[i];
        count = count + 1;
    return count;

class SmallUnderLarge(Set):
  def union(self,i,j):
    i = self.find(i)
    j = self.find(j)
    if i == j:
      return False
    if self.g[i] < self.g[j]:
      i,j = j,i
    self.a[j] = i # j under i
    if self.g[i] == self.g[j]:
      self.g[i] = self.g[i] + 1
    return True

class ConsolidateFind(Set):
  def find(self,i):
    root = i
    while self.a[root] != root:
      root = self.a[root]
    while self.a[i] != root:
      next = self.a[i]
      self.a[i] = root
      i = next
    return root

  # recursive version
  def find_recursive(self,i):
    if self.a[i] == i:
      return i
    self.a[i] = self.find(self.a[i])
    return self.a[i]
```

11.4. Beispiel: Sliding Window

```python
def main():
  text = input()

  map = {'a':0, 'b':0, 'c':0}
  bestl = -1
  bestr = len(text)
  l=0
  r=-1
  num=0
  while r < len(text):
    if num == 3 and bestr-bestl > r-l:
      bestl = l
      bestr = r
    if num >= 3:
      x = text[l]
      if x in map:
        xc = map[x]
        xc -= 1
        map[x]=xc
        if xc == 0:
            num -= 1
      l += 1
    else:
      r += 1
      if r < len(text):
        x = text[r]
        if x in map:
            xc = map[x]
            xc += 1
            map[x] = xc
            if xc == 1:
              num += 1
  if bestl == -1:
    print(text,"does not contain a,b AND c.")
  else:
    print("contains a,b,c between",bestl,"and",bestr)
```

11.5. Beispiel: Palindrome Checker

```python
def isPalindrome(word):
  for i in range(0, len(word)//2):
    if word[i] != word[-1 -i]:
      return False
  return True

def main():
  again, word = True, input("Enter a word: ")
  while again:
    if isPalindrome(word):
      cprint(word + ' is a palindrome!')
    else:
      cprint(word + ' is not a palindrome')
    word = input("Enter a word (or just <ENTER> to stop): ")
    again = len(word) > 0
```

11.6. Dictionary in C++

```cpp
#include <unordered_map>

// Create an unordered_map of strings that map to strings
std::unordered_map<std::string, std::string> colours = {
  {"RED", "#FF0000"}, {"GREEN", "#00FF00"}
};
colours["BLUE"] = "#0000FF"; //Add element
```

```cpp
std::cout << "hex of red: " << colours["RED"] << "\n";


auto search = colours.find("BLUE"); //iterator to object
if (search != colours.end()) {
    std::cout << "Found " << search->first << " : " << search->second << '\n';
} else {
    std::cout << "Not found\n";
}

//iterate
for (const auto& entry : colours) {
  std::cout << entry.first << " : " << entry.second << ", ";
  //BLUE : #0000FF, RED : #FF0000, GREEN : #00FF00,
}
```

12. Anhang

12.1. Nützliche Formeln für asymptotische Laufzeiten

Gauss'sche Summenformel

$$\boxed{\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2} \in \theta\left(n^2\right)}$$

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{\overbrace{n \cdot (n-1) \cdots (n-k)!}^{k\text{-Faktoren}}}{k! \cdot (n-k)!} \in \theta\left(n^k\right)$$

Spezielle Summen

$$\sum_{i=0}^{10n} \log n^n \in \theta\left(10 \cdot n \cdot \log n^n\right) \in \theta\left(n^2 \cdot \log n\right)$$

12.2. Asymptotische Laufzeiten C++

	Wahlfreier Zugriff	Einfügen	Iteration nächstes
std::vector	$\Theta(1)$	$\Theta(1)$ A	$\Theta(1)$
std::list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
std::set	-	$\Theta(\log(n))$	$\Theta(\log(n))$
std::unordered_set	-	$\Theta(1)$ P	-

	Einf. nach Element	Suchen (x in S)
std::vector	$\Theta(n)$	$\Theta(n)$
std::list	$\Theta(1)$	$\Theta(n)$
std::set	-	$\Theta(\log(n))$
std::unordered_set	-	$\Theta(1)$ P

A: Amortisiert
P: Erwartet
sonst: worst case