Verbundstrukturen der Standartbibliothek

Strings

- Direkte Behandlung von Teilstrings mittels Operatoren möglich (str=str+“Neuer String“)

Vectoren

- wird keine Größe angegeben zunächst keine Elemente
- Änderung der Größe mittels resize
- Kopieren der Vektoren möglich
- Überprüfung möglich (try{ ivec.at(2); } catch(out_of_range)…

Listenstrukturen (deque)

- Im Gegensatz zu Vector kein wahlfreier Zugriff auf Daten
- Funktionen: push_front(ele), push_back(ele), pop_front(), pop_back(), front(), back()
- Zum Durchlaufen einer Liste bzw. zum Bearbeiten oder Ändern einer Liste werden Iteratoren verwendet. Ein Iterator kann als spezieller Verweis auf ein Listenelement verstanden werden
  Beispiel:
  Deque<int>::iterator pdeque;
  For(pdeque=deq.begin(), ii=0; pdeque!=deq.end(); pdeque++, ii++)
  *Cout<<‘‘[“<<ii<<“]=“<<*pdeque<<endl;*

Stringtabellen (Maps)

- Handhabung von Schlüssel / Wert-Paren (zu jedem Schlüssel darf höchstens ein Wert vorhanden sein)
- Schlüssel: Schlüsseltyp **first**;
- Datum: Datentyp **second**;

Sortierverfahren

Fragen

- Welche Sortieralgorithmen gibt es?
- Warum und wieso gibt es verschiedene?

Insertionsort

- Aufwand: n^2
- an intuitives Verhalten von Menschen bei Sortierung angelehnt
- noch nicht sortiertes Element wird genommen
- Überprüfung ob linker Nachbar des Elements größer ist:
  1. wenn ja, dann linker Nachbar nach rechts schieben und Element mit neuem linken Nachbarn vergleichen
  2. wenn nein, Element stehen lassen und nächstes unsortiertes Element auswählen

Quicksort

- Gutes Laufzeitverhalten $n \cdot \log(n)$
- rekursive (absteigende) Struktur
- besonders effektiv wenn immer in der genau in der Mitte geteilt wird (ineffektiv bei sortierten Arrays)
- 'median' Suchen – ungefähr mittlerer Wert
- alle Elemente die kleiner sind als Median werden links von Median einsortiert
- alle Elemente die größer sind werden rechts von Median einsortiert
- dadurch Aufteilung in Untergruppen
- dann erneut 'median' suchen und sortieren
- Abbruch wenn nur noch Vektoren mit einem oder keinem Element vorhanden sind
  → keine weitere Sortierung notwendig

Sortieren mit linearem Aufwand (Linsort)

- Aufwand linear
- Bsp. Postleitzahlen
- es wird mit letzter Ziffer begonnen → Postleitzahlen werden in Stapel abhängig von ihrer letzten Ziffer unterteilt
- anschließend Stapel aufsteigend untereinander zurückgestapelt und mit nächster Ziffer fortfahren
- es müssen zu keiner Zeit zwei PLZ miteinander verglichen werden
- es werden n Durchläufe benötigt, in jedem Durchlauf wird jede PLZ einmal verarbeitet
  - ➔ Für große Datenmengen den anderen Sortierverfahren weit überlegen (auf Kosten von Speicherplatz)
  - ➔ Nicht allgemein implementierbar, da konkrete Informationen über die Schlüssel (Anzahl Stellen, vorkommende Ziffern) benötigt werden
- Problem: es wird zusätzlicher Speicherplatz benötigt u. ist nicht immer anwendbar

Topologisches Sortieren (Top-Sort)

- Verwendung bei flexiblen, dynamischen Datenstrukturen
- gewisse Paare sind vergleichbar aber nicht alle (Bsp. Studienplan)
- Sortierung nach Relationen → 3 Axiome

  Bsp: ***x,y,z*** *→ Werte der Menge* ***S***

  1.) *Transitivität →* ***x<y*** *und* ***y<z*** *dann* ***x<z***
  2.) *Asymmetrie →wenn* ***x<y*** *dann nicht* ***y<x***
  3.) *Irreflexivität →nicht* ***x<x***

  ➔ Diese Eigenschaften garantieren, dass der Graph keine Schleifen enthält

- Ablauf der Sortierung
  1. Listenelement mit keinem Vorgänger wird gesucht (muss existieren da sonst Schleife)
  2. Dieses Element kommt an den Kopf der Liste und wird aus der Menge S entfernt
  3. Verfahren bis die Menge S leer ist
- Zur Implementierung sollte jedes Element durch folgende Komponenten dargestellt werden:
  - Identifikationsschlüssel
  - Menge seiner Nachfolger
  - Zähler seiner Vorgänger

Leistungsanalyse von Algorithmen

Die Idealvorstellung, dass eine Laufzeitfunktion immer (d.h. für alle Funktionswerte) besser ist als eine andere, wird sich im Allgemeinen nicht ergeben. Die Wahl des richtigen Algorithmus muss daher in Abhängigkeit der zu erwartenden Funktionswerte (Typ, Menge) getroffen werden.

exponentiell	-abgestuft nach Größe der Basis -inakzeptabel wachsender Zeitbedarf -vermeiden wo immer es möglich ist
polynominal	-abgestuft nach höchster vorkommender Potenz -akzeptabel wachsender Zeitbedarf -höchste Potenz so niedrig wie möglich halten
logarithmisch	-gleichwertig, unabhängig von der Basis -sehr moderates Wachstum -wünschenswert
konstant	-beste Laufzeiteigenschaften -kommt in der Regel aber nicht vor

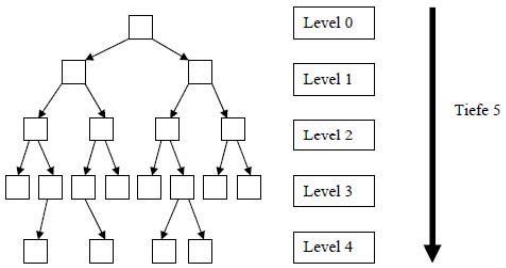
Bäume

Durch baumartige Strukturen lässt sich der Suchaufwand von einzelnen Elementen erheblich reduzieren.

<u>Sortierte Bäume</u>

Aufsteigend Sortiert:
Für jeden Knoten **K** gilt, **X<K** für alle Knoten **X** im **linken** Teilbaum des Knoten **K** und **X>K** für alle Knoten **X** im **rechten** Teilbaum des Knoten **K**

Absteigend Sortiert:
Für jeden Knoten **K** gilt, **X<K** für alle Knoten **X** im **rechten** Teilbaum des Knoten **K** und **X>K** für alle Knoten **X** im **linken** Teilbaum des Knoten **K**



Zum Auffinden eines Elements werden maximal so viele Schritte benötigt wie der Baum tief ist.

Traversierung von Bäumen

Unter TRAVERSIERUNG eines Baumes wird das systematische Aufsuchen aller Knoten des Baumes, zur Durchführung bestimmter Operationen verstanden.

Gemeinsamkeiten: zuerst linken, dann rechten Zweig besuchen
Unterschied: Position an der action()-Funktion ausgeführt wird

Preorder Traversierung

- In der Hierarchie absteigend wird immer ein Knoten besucht
- Die notwendige Bearbeitung durchgeführt
- Zunächst linker dann rechter Teilbaum unterhalb des Knotens

Code:

```
if( n)
{
    action( n);
    preorder( n->left);
    preorder( n->right);
}
```

Inorder-Traversierung (z.B. sortierte Folge)

- Bearbeitung zwischen Traversierung von linkem u. rechtem Teilbaum

Code:

```
if( n)
{
    inorder( n->left);
    action( n);
    inorder( n->right);
}
```

Postorder-Traversierung (zum Löschen eines Baums sinnvoll)

- Zunächst werden angehängte Teilbäume der Knoten traversiert (links dann rechts)
- Anschließend der Knoten selber

Code:

```
if( n)
{
    postorder( n->left);
    postorder( n->right);
    action( n);
}
```

Level-Order-Traversierung

- Abarbeitung des Baumes Ebene für Ebene
- Zuerst linke Seite, dann rechte Seite eine Ebene bearbeiten

Huffmann Codierung

Erzeugt optimalen Präfixcode (kein besserer Baum kann gefunden werden)

➔ Codekomprimierung

Beispiel:
- wenig genutzte Buchstaben bekommen langen Code
- viel genutzte Buchstaben bekommen kurzen Code

- Jedem Blatt ist die relative Häufigkeit w_i eines Zeichens zugeordnet
- Optimierung durch Anordnen der Knoten von oben nach unten mit absteigenden Gewichten
  ➔ Knoten mit großen Gewichten haben kurze Wege
  ➔ Knoten mit kleinen Gewichten haben lange Wege

Graphen

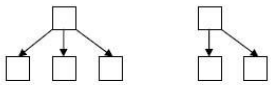
Graphen werden verwendet um eine Beziehung zwischen Objekten darzustellen und eine Bewertung dieser Beziehungen vorzunehmen.

Unterschied Baum – Graph

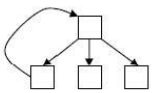
- Baum keine schleifen -eindeutiger Weg zum Blatt
- Graph hat schleifen

Baum: Jeder Knoten in einem Baum lässt sich von der Wurzel aus auf genau einem Weg erreichen, daher sind folgende Kombinationen nicht möglich:

Kein Zusammenhang

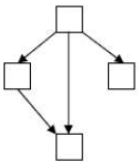


Zyklische Verkettung



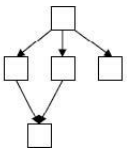
Bei Graphen möglich

Überspringen von Knoten



Bei Graphen möglich

Verschmelzen von Teilbäumen



Bei Graphen möglich

Eigenschaften von Graphen

- Ein Graph der unzerlegbar und Schleifenfrei ist heißt **Baum**
- Ein Graph heißt **Wurzelbaum**, wenn es von einem Knoten w(Wurzel) zu jedem Knoten k des Graphen genau einen Weg gibt.

Gerichteter Graph	Eine Kante (a,b) verbindet den Knoten A mit Knoten B, aber nicht umgekehrt A → Startknoten B → Endknoten
Ungerichteter Graph (Symmetrischer Graph)	Eine Kante (a,b) verbindet sowohl den Knoten A mit dem Knoten B, als auch den Knoten B mit dem Knoten A

<u>Wege in Graphen</u>

Geschlossen, Schleife	Anfangs-u. Endknoten des Weges sind gleich
Schleifenfrei	Alle Knoten des Weges sind voneinander verschieden
Kantenzug	Alle zur Verbindung der Knoten verwendeten Kanten sind voneinander verschieden
Geschlossener Kantenzug	Kreis

Schwach zusammenhängend:
Alle Knoten sind irgendwie verbunden aber nicht jeder Knoten kann von jedem anderen Knoten erreicht werden.

Stark zusammenhängend:
Alle Knoten sind miteinander verbunden und jeder Knoten ist von jedem anderen beliebigen Knoten erreichbar.

Bewerteter Graph: Ein Graph mit zugeordneten Kostenfunktionen

Darstellung von Graphen in Datenstrukturen

Adjazenzmatrix	• Matrix mit Dimension = Anzahl Knoten • Gibt es zwischen zwei Knoten eine Verbindung wird eine 1 eingetragen • Gibt es zwischen zwei Knoten keine Verbindung wird eine 0 eingetragen • Sollen auch die Werte von Kostenfunktionen beachtet werden so gelten Werte größer 0 als existierende Verbindung (Kosten-Wege-Matrix) • Nur rentabel wenn Kantenanzahl zu Knotenanzahl nicht zu gering • Speicherplatz: n^2
Adjazenzliste	• Zu jedem Knoten wird eine Liste gespeichert, die seine unmittelbaren Nachfolgerknoten enthält • Liste mit Verweis auf Liste (sehr dynamisch) • Vector mit Verweis auf Liste (mittlere Flexibilität) • Zwei Vektoren (geringe Flexibilität) • Rentabel wenn Kantenanzahl zu Knotenanzahl gering • Speicherplatz: n+m
Inzidenzmatrix	• Matrix mit Knoten als Zeilen und Kanten als Spalten • Ist ein Knoten ein Startpunkt einer Kante wird eine 1 eingetragen • Ist ein Knoten ein Endpunkt einer Kante wird eine -1 eingetragen • Erlaubt die Darstellung von Parallelkanten aber keine Schleifen • Speicherplatz: n*m
Kantentabelle	• Matrix (Tabelle) mit Kanten als Spalten und Start- und Endknoten als Zeile • Anfangs- und Endknoten jeder Kante wird in der Tabelle Festgehalten • Eignet sich zur Darstellung von Graphen mit Parallelkanten und Schingen • Speichplatz: 2*m
Wegematrix	• Matrix der Dimension = Anzahl Knoten • Gibt es einen Weg von Knoten A nach Knoten B wird eine 1 eingetragen, sonst eine 0 • Wegematrix muss aus Adjazenzmatrix gebildet werden können • Multiplikation der Adjazenzmatrix mit sich selbst ergibt Wegematrix

Traversierung von Graphen

Traversierung von Graphen → führt zu einem aufspannenden Baum

Vorgehensweise:
Es werden Stück für Stück alle Knoten besucht und als besucht markiert. Kommt man wieder zum Ausgangsknoten oder ist ein Durchgang abgeschlossen und es wurden noch nicht alle Knoten besucht, so wird der nächste nicht besuchte Knoten gesucht und dort fortgefahren. Wenn alle Knoten einmal besucht wurden ist die Traversierung abgeschlossen.

Algorithmen zur Bestimmung des kürzesten Weges in Graphen

Aufgabe 1: Finde und beschreibe den kürzesten Weg zwischen zwei Knoten A und B
Aufgabe 2: Finde und beschreibe die kürzesten Wege von einem festen Knoten A aus, zu allen anderen Knoten des Graphen.
Aufgabe 3: Finde und beschreibe die kürzesten Wege für alle Knotenpaare in einem Graphen

Die Lösung der Aufgabe 1 beinhaltet immer die Lösung der Aufgabe 2, da beim Suchen des kürzesten Wegs zwischen A und B auch alle anderen kürzesten Wege als Abfallprodukt anfallen.

→ Nur Aufgabe 2 und 3 relevant

Floyd	• Löst Aufgabe 3 • Knotenorientiert • Für Navigation in Straßennetzen geeignet • Es wird immer ein Zwischenknoten herausgepickt und auf alle Kombinationen der Start- und Endknoten untersucht. Wenn er besser ist als der Alte wird er eingespeichert • Aufwand: n^3 (n=Knoten im Graphen)
Dijkstra	• Löst Aufgabe 2 • Knotenorientiert • Ergebnis wird ein Wurzel-Baum sein • Datenstruktur: Adjazenzmatrix • Es wird eine Menge von „erledigten“ Knoten aufgebaut, die in jedem Verfahrensschritt um ein Element vergrößert wird. Man hangelt sich so immer um einen Knoten weiter • Laufzeit unabhängig von der Anzahl der Kanten • Gut geeignet für dicht besetzte Adjazenzmatrizen • Aufwand: n^2
Ford	• Löst Aufgabe 2 • Kantenorientiert • Datenstruktur: Kantentabelle • Es werden alle Kanten durchlaufen. Wenn sich ein kürzerer Weg ergibt so wird dieser in den Baum eingebaut und die alte Kante gelöscht • Laufzeit abhängig von Anzahl der Kanten ➔ Gut geeignet bei wenig Kanten im Verhältnis zu Knoten ➔ Gut geeignet für Straßennetz • Aufwand: n*m

Minimal aufspannende Bäume

- Spannbaum: von der Anzahl der Kanten her nicht verkleinerbarer zusammenhängender Teilgraph
- erreicht alle Knoten des ursprünglichen Graphen

Spannbäumen minimaler Länge:
Anwendung: in vorhandenen Datennetz, Städte mit möglichst kurzen Kabeln zu verbinden

➔ Algorithmus von Kruskal