Zusammenfassung GGIF 1

Inhalt

1. Elemente der Sprache C (vor allem Syntax)
- **Datentypen:**
  - Elementare Datentypen
  - Abstrakte Datentypen
  - Deklaration von Variablen und Konstanten
  - Zeiger
  - Zusammengesetzte Datentypen
    - Arrays
    - Strukturen
  - type casting
- **Anweisungen**
  - Arithmetische und logische Operationen
  - Kontrollanweisungen
    - Verzweigungen
      - if
      - switch
    - Schleifen
      - for
      - while
      - do-while
- **Programmaufbau**
  - Funktionen
    - lokale & globale Variablen
    - call by value & call by reference
  - Module

2. Analyse von Algorithmen
- **O-Notation**
  - Definition
  - Rechenregeln
  - Wichtige Wachstumsordnungen
- **Analyse**

3. Sortieralgorithmen
- **Einfache Verfahren**
  - Selection Sort
  - Insertion Sort
  - Bubblesort
- **Quicksort**
- **Heapsort**

Jeweils: Ablauf, Funktionen & Laufzeiten, Stabilität!

4. Lineare Datentypen

- **Listen**
  - Aufbau
  - Operationen & Laufzeiten
- **Stacks (LIFO)**
  - Operationen: push & pop
  - Implementierungen: Listen & Arrays
- **Queues (FIFO)**
  - Operationen: put & get
  - Implementierungen: Listen & Arrays

5. Bäume

- **Definition, Binärbäume, Implementierung**
- **Baumdurchläufe: Ablauf & Implementierung von...**
  - ...Preorder
  - ...Inorder
  - ...Postorder
- **Suchbäume**
  - Definition
  - Operationen & Laufzeiten

6. Graphen

- **Definition & Eigenschaften**
  - Pfade
  - gerichtet
  - zusammenhängend
  - zyklenfrei
  - Baum
  - gewichtet
- **Dijkstra-Algorithmus**
  - Definitionen & Mengen
  - Ablauf
  - Implementierung
- **Repräsentationen: Eigenschaften & Implementierung von...**
  - ...Adjazenzmatrix
  - ...Adjazenzlisten
- **Graphdurchläufe: Implementierung & Laufzeiten von**
  - DFS
  - BFS
- **Kruskal-Algorithmus**
  - Ablauf
  - Implementierung
  - Laufzeit

7. Techniken zum Algorithmenentwurf

- **Rekursion & Iteration**
  - Ablauf
  - Laufzeitanalyse

→ ab hier nicht mehr in der Zusammenfassung enthalten!

- **Backtracking**
  - Idee
  - Ablauf
- **Kombinatorische Optimierung**
  - Greedy-Algorithmen
- **Schwierige Probleme & Heuristiken**
  - NP-vollständige Probleme
  - Heuristiken
- **Dynamische Programmierung**
- **Branch & Bound**
  - Idee
  - Schranken

1. Elemente der Sprache C

Datentypen

Elementare Datentypen

- char: Zeichen, 8 Bit
- int: Ganzzahlen, mind. 16 Bit (short int) bzw. mind. 32 Bit (long int), signed / unsigned
- float: Fließkommazahl, double: größerer Wertebereich, long double: noch größerer Wertebereich
- benutzerdefinierter int-Typ: enum ID {Wert1=x, Wert2=y}, Wert1 und Wert2 als symbolische Konstanten für x und y, Standard-Aufzählungsbeginn ist 0

Abstrakte Datentypen

- ADT nur über Schnittstelle zugänglich, Implementierung bleibt verborgen
- In C: Daten und Operationen getrennt, daher keine tatsächlichen ADT möglich

Deklaration von Variablen und Konstanten

- Deklaration von Variablen vor 1. Verwendung → bestimmt benötigten Speicherplatz, ermöglicht Korrektheitsprüfungen durch Compiler
- Deklarationsmuster: Typ ID (, ID)* ;
- ID als regulärer Ausdruck: ID = L(L|D)* (L=Letter, D=Digit)
- typedef zur Abkürzung langer Typen, Beispiel: typedef unsigned long int UL; (UL als Abkürzung für unsigned long int)
- symbolische Konstanten (Textersetzung): #define KONSTANTE (ohne Semikolon!)
- konstante Variablen: const Typ ID = Wert; (danach keine weitere Zuweisung an a!)

Zeiger

- allg. Definition: Variable, die Speicheradresse einer Variablen enthält
- auch Zeiger sind Objekte im Speicher, auf die andere Zeiger zeigen können
- Deklarationsmuster: Typ * ID (* gehört syntaktisch zur ID)
- Dereferenzierung: *-Operator, Bespiel: int *x, y; y = *x (weist y den Wert von x zu), * nur auf Zeiger anwenden (andere Objekte sind keine Adressen!, Ausnahme int)

- Erzeugung von Referenzen: &-Operator, Beispiel: int *x, y; x = &y (weist dem Zeiger x die Adresse von y zu), & nur auf Objekte im Speicher anwenden (Adressen sind selbst keine Objekte mehr, Zeiger allerdings schon)
- & und * sind invers zueinander: *(&x) = x
- Zwecke von Zeigern:
  - Zugriff auf mehrere Objekte mittels eines Objekts (des Zeigers)
  - Zuweisung von dynamischem Speicher (Heap):
    - Zuweisung mit malloc(), Bsp: int *p = (int*) malloc(1000); (für int-Objekt mit 1000 Bytes), gibt Anfangsadresse eines freien 1000-Byte-Blocks zurück
    - Freigeben des Speichers mit free(), Bsp: free(p);

<u>Zusammengesetzte Datentypen</u>

Arrays:
- Definition: Felder bzw. Vektoren von Variablen gleichen Typs
- Deklaration: Typ ID [int-Konstante], Zugriff: ID[index]
- Indizierung beginnt in C bei 0 → bei Deklaration von n Elementen ist n-1 der größte Index
- Array-Objekte im Speicher hintereinander, Index = Offset von Basisadresse, Bsp: int arr[10]; → arr[i] = *(&arr[0] + i) (&arr[0] = arr ist Basisadresse)
- wegen &arr[0] = arr können Arrays nicht direkt kopiert werden (&arr[0] = &arr2[0] ungültig) → nur elementweises Kopieren per Schleife möglich
- Datentyp String: Darstellung als Char-Array, letztes Byte enthält immer '\0' → Deklaration von Strings immer ein Byte größer als gewünschte Länge!
- Mehrdimensionale Arrays möglich (doppelte Indizierung: arr[x][y])

Strukturen:
- Zusammenfassung von Variablen unterschiedlichen Typs
- Adressierung per Namen, nicht per Index
- Deklarationsmuster: struct ID {Variablendeklaration*}, Variablen können auch wieder structs sein!
- Oft in Verbindung mit typedef, Bsp. struct point {float x,y;} p1, p2;
  = typedef struct point {float x,y;} Point; Point p1, p2;
  (Point als Abkürzung für struct point)
- Kopieren: p1 = p2; (kopiert alle Komponenten auf einmal, kann für Kopieren von Arrays benutzt werden)
- Zugriff auf Komponenten mit .-Operator, Bsp: p1.x = 20.2; p2.y = p1.x;
- bei Zeigern auf structs Zugriff auch mit ->-Operator möglich,
  Bsp: Point *p_ptr; p_ptr->x = 13.4; (alternativ mit .-Operator: (*p_ptr).x = 13.4;)

<u>Type Casting</u>
- Implizite type casts von elementaren Datentypen: Bei Berechnung von Ausdrücken mit gemischten Typen (z.B. int- & float-Variablen) wird in den komlexeren gecastet (int in float), das Ergebnis ist vom komplexeren Typ (float-Ergebnis)
- bei Zuweisung an int-Variable wird implizit wieder zurückgecastet,
  Bsp: int a, b = 10; float c = 3.4; a = b + c; (cast von b in float, cast von float-Ergebnis in int)
- Type Casts nicht komplett standardisiert, compilerabhängig!
- Explizite type casts: Ziel = (Typ) Ausdruck; (Ausdruck wird in Typ gecastet und Ziel zugewiesen), z.B. bei malloc() notwendig, da malloc() immer void-Zeiger zurückgibt

Anweisungen

Arithmetische und logische Operationen

Arithmetische Operationen:

- Rechenoperationen +, -, *, / (bei zwei int-Operanden auch int-Ergebnis, Bsp: 3/5 = 0, 3./5 = 0.6), % (nur bei Ganzzahlen)
- Punkt-vor-Strich-Rechnung
- Mathematische Funktionen in math.h (Standard-Library)
- Shift-Operationen << (Linksshift, x<<1 = x*2), >> (Rechtsshift, x>>1 = x/2): bitweise Verschiebung, Rechtsshift bei signed-Typen: links mit Vorzeichen auffüllen, bei unsigned-Typen: links mit Nullen auffüllen

Logische Operationen:

- Vergleichsoperationen <, >, <=, >=, ==, !=, Ergebnis ist true oder false (in C integer-Werte statt Boolean: true = von 0 verschieden (z.B. 1), false = 0)
- Logische bitweise Verknüpfungen: &(UND), |(ODER), ^(XODER)

Kontrollanweisungen

Verzweigungen:

- if-then-else, Syntax: if (Bedingung) {statement*}
  bzw. if (Bedingung) {statement*} else {statement*},
  Bedingung ist in C vom Typ int (wieder false = 0), Verknüpfung von Bedingungen mit && (UND), || (ODER) → Short-Circuit-Auswertung (Teilausdrücke nach erstem false (bei UND) bzw. nach erstem true (bei ODER) werden nicht mehr geprüft, da irrelevant), Dangling-Else-Problem (Konvention: else gehört zum innersten freien if, Umgehen durch Benutzung von Blockstruktur {})
- switch, Syntax:
  switch (Ausdruck) {case Konstante1: statement* break;
  case Konstante2: statement* break;
  default: statement* break;},
  ohne break-Statements Abarbeitung aller cases und evtl. unerwünschtes Verhalten, default-Zweig wird ausgeführt, wenn kein case zutrifft, switch-Anweisung ermöglicht effizienteren Assembler-Code als gleichwertige if-Statements

Schleifen:

- Wiederholtes Ausführen eines Anweisungsblocks in Abhängigkeit einer Variablen
- alle drei Schleifentypen in C im Prinzip äquivalent
- for-Schleife: wenn Anzahl der Durchläufe feststeht, Syntax:
  for (Initialisierung;Bedingung;post-statement) {statement*},
  Initialisierung(Zuweisung) ist auch ein Ausdruck (Bsp: c = 5 hat als Wert 5)
- do-Schleife: wenn mind. ein Durchlauf stattfinden muss, Syntax:
  do {statement*} while (Bedingung), Bedingung wird nach Ausführung der statements geprüft
- while-Schleife: sonst, Syntax: while (Bedingung) {statement*}
- Statements für Ausnahmefälle: continue; (beendet laufende Iteration), break; (Verlassen der kompletten Schleife)

Programmaufbau

<u>Funktionen</u>

- Programm ist zu jedem Zeitpunkt in genau einer Funktion, zu Beginn main()
- Funktionen können Unterfunktionen aufrufen u. Übergeben Parameter an diese, nach Ausführung Rücksprung ins Programm hinter den Funktionsaufruf
- Rekursion: Selbstaufruf einer Funktion
- jede Funktion besitzt Deklaration und Definition, Deklaration oder Definition muss im Code vor erstem Aufruf der Funktion stehen
- Deklaration (Interface): Name, Definitionsbereich, Wertebereich, Deklarationsmuster: Typ ID (Parameter-Deklaration*) (Parameter ohne ID möglich)
- Definition (Implementierung): Programmcode,
Definitionsmuster: Typ ID (Parameter-Deklaration*) {statement*},
return-Statement zum Verlassen der Funktion, Syntax: return Ausdruck,
Ausdruck muss vom gleichen Typ wie die Funktion sein (falls nötig: type cast!),
bei Funktionstyp void: kein Rückgabewert, kein return-statement nötig
- bei Aufruf einer Funktion: Formale Parameter des callee (aufgerufene Funktion) werden durch aktuelle (vom caller übergebene) Parameter ersetzt
- globale Variablen: im gesamten Programm gültig, möglichst wenig benutzen!
- lokale Variablen & formale Parameter: Deklaration innerhalb der Funktion, für die sie gültig sind, verdecken gleichnamige globale Variablen
- call by value: Kopie des Parameterobjekts wird übergeben (also nur der Wert), Objekt selbst kann nicht verändert werden
- call by reference: Referenz auf das Objekt wird übergeben, Objekt dadurch veränderbar (nur C++!), in C nur call by value mit Zeigern möglich
- bei structs: Übergabe von Adresse zeit- und platzsparender als Übergabe der ganzen struct
- Funktionen besitzen Speicheradressen, also kann man Zeiger auf Funktionen definieren und als Parameter an allg. Funktionen übergeben,
Bsp:

```
int add(int a,int b)
{ return a+b; }
int sub(int a,int b)
{ return a-b; }

int op(int (*f)(int,int),
 int a, int b)
{ return f(a,b);
}

main()
{ char c; int result;
  scanf("%c",&c);
  if (c == 'a')
    result = op(&add,1,2);
  else if (c == 's')
    result = op(&sub,1,2);
}
```

Module

- C-Programm kann aus mehreren Modulen bestehen, die einzeln bearbeitet und vom Compiler übersetzt werden, Module werden vom Linker zu einem Programm zusammengesetzt
- Modul = Globale Variablen und Funktionen, Modul kann aus Interfaceteil (enthält Schnittstellen / Deklarationen, für andere Module sichtbar) und Implementations-Teil (für andere Module verborgen) bestehen
- ein Modul muss keine main-Funktion enthalten, aber ganzes Programm muss genau eine main-Funktion enthalten!
- globale Variablen und Funktionen gleichen Namens nur einmal im ganzen Programm erlaubt!

2. Analyse von Algorithmen

Analyse

- Komplexitätsanalyse in Abhängigkeit der Eigabegröße oder -menge zur Abschätzung von Laufzeit und/oder Speicherbedarf
- Analysemöglichkeiten:
  1. Zählen der Einzelschritte, aber aufwendig, rechnerabhängig
  2. Abstrakte Analyse: z.B. Anzahl Vergleiche, Anzahl zu vertauschender Paare; rechnerunabhängig, aber grob
- Welcher Algorithmus ist schneller für bestimmte Eingabegröße (n)? → Für große n der Alg., dessen Laufzeit weniger mit n steigt
- Allgemeine Analyse: Frage: Wie wächst die Laufzeit mit der Eingabegröße? → Konstante Faktoren vernachlässigen (durch Rechnerunabhängigkeit wenig aussagekräftig, bei großen n vernachläsigbar)

O-Notation

Definition

- Def.: Sind $f : N \to N$ und $g : N \to N$ zwei Funktionen, so ist
  $f \in O(g)$, falls
  $\exists\, n0 \in N$, $\exists\, c \in R$, so dass
  $\forall\, n \geq n0$: $f(n) \leq c \cdot g(n)$,
  → $f \in O(g)$ heißt: f wächst nicht schneller als g
- O(g) ist die Menge aller Funktionen, die höchstens so schnell wachsen wie c*g ,
  O: Wachstumsordnung von g
- Konstanten werden vernachlässigt, am stärksten ansteigender Term setzt sich durch
- Bsp: 10^{20} n2 $\in$ O(1/100 n3) nicht sinnvoll, in der Praxis aber kleine konst. Faktoren

Rechenregeln

- „O“ im Ausdruck vernachlässigen
- Umformen des Ausdrucks wie gewohnt
- Nur den führenden Term (höchster Exponent) behalten
- Addition: $f+g \in O(\max\{f,g\})$
- Multiplikation: $f*g \in O(f*g)$
- Linearität: $f(n) = c1g(n) + c2$ mit c1 und c2 aus R, so ist $f \in O(g)$

wichtige Wachstumsordnungen

- O(1), O(log2(n)), O($\sqrt{(n)}$), O(n), O(n*log2(n)
- O(n²), O(n³), O(2^n)

Suche in Arrays (n Elemente)

- Lineare Suche: worst case & average case O(n)
- Binäre Suche: Ann.: Array ist aufsteigend sortiert, Vergleich des Suchschlüssels (key) mit „mittlerem“ Array-Element A[k]
→ Suche in linker Hälfte, wenn key <= A[k]
→ Suche in rechter Hälfte, wenn key > A[k]
Fortsetzung bis key gefunden oder keine Teilung mehr möglich,
bei Implementierung: keine wirkliche Teilung, nur Umsetzen der Suchgrenzen-Indizes (z.B. l & r), Suche solange wie r >= l (sonst keine Teilung mehr möglich),
worst case & average case O(log n) (n-elementiges Array kann log n -mal halbiert werden)

3. Sortieralgorithmen

Definition des Sortierproblems

- Gegeben: Folge S = (s1,...,sn) von Datensätzen (structs)
- Jeder struct si besitzt eine Komponente si.key eines mittels einer Ordnungsrelation linear geordneten Datentyps
- Gesucht: Permutation (Anordnung) S´ = (si1,...sin) der Elemente von S, so dass si1.key ≤ si2.key ≤ ... ≤ sin.key
- wenn ursprüngliche Reihenfolge bei gleichem Schlüssel beibehalten wird, heißt ein Sortieralgorithmus „stabil“
- Laufzeitanalyse: Vergleiche und Vertauschungen wesentlich, Beschränkung auf Vergleiche da obere Schranke für Vertauschungen
- man weiß: untere Schranke für Sortieralgorithmen im worst case ist O(n*log n)

Einfache Verfahren

- links entsteht sortiertes Teilarray
- Laufzeit: O(n²)

Selection Sort

- Durchlaufen des Arrays von links nach rechts, Suchen des kleinsten Elements, Vertauschen des Minimums mit erstem Array-Element, Fortsetzung mit unsortiertem Teilarray
- Minimum zunächst auf jeweils erstes Element des unsortierten Teilarrays setzen
- beim Durchlaufen: Vergleich des aktuellen Elements mit bisherigem Minimum
- so lange, bis unsortierter Teil leer ist
- Laufzeit: O(n²), auch bei vorsortiertem Array (n-1 Iterationen der äußeren Schleife, max. n-1 Vergleiche in innerer Schleife)
- nicht stabil!

Insertion Sort

- Durchlaufen des Arrays v.l.n.r., Elemente links von aktuellem Element i so lange nach rechts rücken, bis A[i] >= eins der Elemente links von i, i an dessen Pos. einfügen, Fortsetzung an Pos. i+1, so lange bis i einmal an jeder Position war
- Laufzeit: $O(n^2)$ (n-1 mal äußere Schleife, max. n-1 mal innere Schleife)
- stabil!

Bubble Sort

- Sortieren durch fortgesetztes paarweises Vertauschen von rechts nach links, kleinstes Element wird zunächst nach ganz links getauscht, Fortsetzung mit Array rechts davon
- Vorsortierung des restlichen Arrays mit jedem Durchlauf
- Laufzeit: $O(n^2)$
- stabil!

Quicksort

- Idee: Teile & Herrsche, Zerlegen des Arrays in zu sortierende Teilarrays, Zusammenfügung (Konkatenation) ergibt sortiertes Array
- Trennelement A[k] wählen, andere Elemente so vertauschen, dass alle Elemente links von k <= A[k], alle rechts von k >= A[k] (alles mit Hilfsfunktion partition), zwei rekursive Aufrufe von Quicksort für die beiden Teilarrays
-
```
int partition(int A[], int l, int r)
{ int i,j,k,v;
k = r; v = A[k];   willkürliches Trennelement
i = l;             starte am linken Rand
j = r-1;           starte am rechten Rand – 1
while (1)          durchlaufen bis Abbruch
{     while (A[i] <= v && i < r) i++;     i läuft bis A[i] > v
      while (A[j] >= v && j >= l) j--;    j läuft bis A[j] < v
      if (i >= j)        aneinander vorbeigelaufen ? → Teilarrays schon richtig
            break;       Abbruch der while-Schleife
      else  exchange(A[i],A[j]);     wenn i links von j → vertauschen
}
exchange(A[i],A[k]);   k einfügen: Tausche k mit i
return i;              Pos. von k = Pos. des Trennelements zurückgeben
}
```
-
```
void quick_sort(int A[], int l, int r)
{ int k;
if (r <= l) return;
k = partition(A,l,r);
quick_sort(A,l,k-1);   Rekursiver Aufruf mit Teilarrays links und rechts von k
quick_sort(A,k+1,r);
}
```
- Laufzeit von Sortierungszustand des Arrays abhängig:
  worst case, wenn Trennelement (ganz rechts) das Maximum des Arrays ist → $O(n)$ Vergleiche mit A[i], bei vorsortiertem Array wird immer nur das Trennelement abgespalten → $O(n)$ Teilungen * $O(n)$ Vergleiche → Laufzeit: $O(n^2)$
  best case: Pro Teilung zusammen $O(n)$ Vergleiche mit A[i] und A[j], Halbierung des Arrays log n -mal möglich → Laufzeit (auch im average case): $O(n*\log n)$
- nicht stabil!

Heapsort

- Idee: Selection Sort mit beschleunigter Minimumsauswahl, dazu nötig: Array mit Heap-Eigenschaften → A[i] <= A[2i] && A[i] <= A[2i+1] (wenn 2i bzw. 2i+1 <= N, also noch im Array enthalten sind), Veranschaulichung als Baum: A[i] hat Söhne A[2i] und A[2i+1] → Minimum des Heaps steht in Wurzel A[1]→ deren Entnahme ist O(1)
- Indizierung des Arrays bei Heapsort von 1 bis N statt von 0 bis N-1!
- Umwandlung des Arrays in Heap, Tausch von A[1] und A[n], Heap A[1] bis A[n-1] wieder reparieren, Tausch von A[1] und A[n-1] usw. bis alles sortiert ist → Sortierung hier in absteigender Reihenfolge!
- Reparatur des Heaps: Einsinken lassen der Wurzel → Vertausche Wurzel mit kleinerem der beiden Söhne, dann mit kleinerem der neuen Söhne usw. bis keine kleineren Söhne mehr da sind → Heap repariert
-
```
void sink(int A[], int k, int N) /* A[0] nicht benutzt */
{ int son; /* speichert kleinsten Sohn falls vorhanden */
while (1) /* durchlaufen bis Abbruch */
      { if (2*k > N)
            break; /* Knoten k hat keinen Sohn */
      if (2*k+1 <= N) /* Knoten k hat 2 Söhne */
            { if (A[2*k] < A[2*k+1]) son = 2*k;
            else son = 2*k+1;
            }
      else son = 2*k; /* 2k <= N, Knoten k hat 1 Sohn */
      if (A[k] > A[son]) /* Vertauschen notwendig? */
            { exchange(A[k],A[son]);
            k = son; /* Knoten k evtl. weiter sinken lassen */
            }
      else break; /* sonst: richtige Pos. für k gefunden */
      }
}
```
-
```
void heap_sort(int A[], int N)
{ int i;
for (i = N/2; i > 0; i--)
      { sink(A,i,N); /* Heap aufbauen */
      }
for (i = N; i > 1; i--)
      { exchange(A[1],A[i]);
      sink(A,1,i-1);
      }
}
```
- Laufzeit:
  worst case: O(n) Iterationen der beiden for-Schleifen → O(n) * Tsink + O(n),
  Tsink: in jedem while-Durchlauf nur O(1) Schritte, da k mit jedem heruntersinken auf 2k oder 2k+1 gesetzt wird (also mind. verdoppelt) max. log n while-Iterationen
  → Gesamtlaufzeit im worst case: O(n*log n)
- nicht stabil!

4. Lineare Datentypen

- zur Darstellung von Mengen von Objekten mit bestimmten Operationen (Einfügen, Entfernen, Suchen von Elementen, Schnitt und Vereinigung zweier Mengen)
- Problem bei Darstellung durch Arrays: Größe muss statisch feststehen (Array nicht sicher groß genug / Speicherplatz-Verschwendung bei sehr großem Array)

Listen

Aufbau

- **Def.**: Eine **(verkettete) Liste** ist entweder leer oder besteht aus einer Referenz auf einen **Knoten**, der ein Element und eine Referenz auf eine verkettete Liste enthält → Liste ist rekursive Datenstruktur
- Muster eines Knotens für Datentyp T: typedef struct node { T* data; struct node* next; } *nodeptr; data zeigt auf Datensatz, next auf nächsten Knoten (Datensatz und nächster Knoten an beliebiger Stelle im Heap-Speicher)
- Listenende / leere Liste durch NULL-Zeiger dargestellt

Operationen

- leere Liste erzeugen:
```
void Init(listptr L)
{ L->first = NULL; L->last = NULL; }
```
- Test auf Leerheit:
```
int IsEmpty(List L)
{ return (L.first == NULL && L.last == NULL); }
```
- Neuen Listenknoten erstellen:
```
nodeptr newnode(T* item)
{nodeptr np;
 np = (nodeptr) malloc(sizeof(Node));
 np->data = item;
 np->next = NULL;
 return np;
}
```
- Einfügen am Listenanfang:
```
void AppendFirst(T* item, listptr L)
{nodeptr np = newnode(item);
 if (isEmpty(*L))
      {L->first = np;
       L->last = np;}
 else
      {np->next = L->first;
       L->first = np;
      }
}
```
- Einfügen am Listenende:
```
void AppendLast(T* item, listptr L)
{nodeptr np = newnode(item);
 if (isEmpty(*L))
      {L->first = np;
       L->last = np;}
 else
      {L->last->next = np;
       L->last = np;
      }
}
```
- Test ob ein Datensatz enthalten ist (Sequentielle Suche)
```
int IsIn (T* item, List L)
{nodeptr np;
 if (IsEmpty(L)) return 0;
 np = L.first;
 while (np != NULL)
      {if (Equal(np->data, item)) return 1;
       np = np->next;
      }
return 0;
```

```
}
```
Laufzeit: Best case O(1), worst case O(n), average case O(n/2) = O(n)

- Einfügen hinter einem bestimmten Element

```
void InsertBehind(T* item1, T* item2, listptr L)
{nodeptr np, newnp;
 if   (!IsIn(item1, *L))
      {printf („Fehler!“); return;
      }
 np = L->first;
 while (np != NULL)
 {if (Equal(np->data, item1))
  {newnp = newnode(item2);
   newnp->next = np->next;
   np->next = newnp;
   if (np == L->last)
      {L->last = newnp;}
   break;
  }
  np = np->next;
 }
}
```

- Löschen eines Listenelements

```
void Delete(T* item, listptr L)
{nodeptr np1, np2;
 if (IsEmpty(*L)) return;
 np1 = L->first;
 if (Equal(np1->data, item))
      {L->first = np1->next;
       if (L->first == NULL)
            {L->last == NULL;}
       free (np1);
       return;
      }
 np2 = np1->next;
 while (np2 != NULL)
      {if (Equal (np2->data, item))
            {np1->next = np2->next;
             if (np2 == L->last)
                  {L->last = np1;}
             free(np2);
             break;
            }
       np1 = np2;
       np2 = np2->next;
      }
}
```

- Zusammenfügen zweier Listen

```
listptr Union(listptr L1, listptr L2)
{if (IsEmpty(*L1)) return L2;
 if (IsEmpty(*L2)) return L1;
 L1->last->next = L2->first;
 L1->last = L2->last;
 return L1;
}
```

- Schnittmenge zweier Listen

```
listptr Intersect(listptr L1, listptr L2)
{nodeptr np;
 listptr L= (listptr)malloc(sizeof(List));
 init(L);
 np = L1->first;
 while (np != NULL)
```

```
 {if (IsIn(np->data, *L2))
      AppendLast(np->data,L);
  np = np->next;
 }
 return L;
}
```

Stacks (LIFO)

- Def.: ADT mit Operationen push & pop

Operationen: push & pop

- push: Ablegen eines Objekts oben auf dem Stack
- pop: Entnahme des obersten Objekts
- Beispiel Laufzeit-Stack: Funktionsaufruf → push, Funktionsende → pop

Implementierungen: Listen & Arrays

durch Listen: Elemente auf Stack → Listenelemente, Push → Einfügen am Listenanfang, Pop → Entfernen des ersten Listenelements

- push:
```
void push (T* item, listptr L)
{AppedFirst (item, L); }
```
- pop:
```
T* pop(listptr L)
{T* item;
 if (IsEmpty(L)) return NULL;
 item = L->first->data;
 Delete(item, L);
 return item;
}
```

durch Arrays: max. Größe festgelegt, Elemente auf Stack → Array-Elemente, Stack-Pointer "top“ zeigt auf nächstes freies Array-Element, push → Einfügen und Erhöhen von top um 1 (und Test auf Überlauf nötig!), pop → Verringern von top um 1 und Entfernen

- `#define MAX 100`
- `typedef struct stk {T* elems[MAX]; int top;} stack;`
- `void init (stack s) {s.top = 0;}`
- `int IsEmpty (stack s) {return (s.top == 0);}`
-
```
void Push (T* item, stack* s)
{if (s->top == MAX)
      {printf(„Stack voll!“); return; }
 s->elems[s->top] = item;
 s->top++;
}
```
-
```
T* Pop (stack* s)
{if (IsEmpty(*s)) return NULL;
 s->top--;
 return s->elems[s->top];
}
```

Bespiel: Auswertung von Rechnungen mit Klammer-Ausdrücken

- Operanden- & Operator-Stack
- „(“ → keine Aktion
- Operator → push auf Operator-Stack
- Operand → push auf Operanden-Stack
- „)“ → 1x pop auf Operator-Stack, 2x pop auf Operanden-Stack, Berechnung, 1x push des Ergebnisses auf Operanden-Stack (für evtl. weitere Ausdrücke)

Queues (FIFO)

- Def.: ADT mit Operationen put & get

<u>Operationen: put & get</u>

- put: Element am Ende der Queue einfügen
- get: Entfernen des zuerst eingefügten Elements

<u>Implementierungen: Listen & Arrays</u>

durch Listen: Queue-Elemente → Listenelemente, put → Einfügen am Listenende, get → Entfernen des ersten Listenelements

- ```
  List L;
  void Put(T* item,listptr L)
  {AppendLast(item,L); }
  ```

- ```
  T* Get(listptr L)
  { T* item;
   if (IsEmpty(*L)) return NULL;
   item = L->first->data;
   Delete(item,L);
   return item;
  }
  ```

durch Arrays: zyklische Adressierung notwendig, weil die ersten Array-Elemente sonst bald nicht mehr benutzt werden, d.h. A[0] als Nachfolger von A[n] definieren (aber Überlauf vermeiden!), first- und last-Zeiger auf freiem Element vor bzw. hinter der Queue, %-Operator zur zyklischen Adressierung

- #define MAX 100
  ```
  typedef struct q
  { T* elems[MAX];
  int first, last; } queue;
  ```
- void Init(queue* q)
  ```
  { q->first = 0; q->last = 1; }
  ```
- void Put(T* item,queue* q)
  ```
  { if (q->last == q->first) overflow();
    else
        { q->elems[q->last] = item;
          q->last = (q->last+1) % MAX;
        }
  }
  ```
- T* Get(queue* q)
  ```
  { q->first = (q->first+1) % MAX;
    if (q->first == q->last) underflow();
    else return q->elems[q->first];
  }
  ```

5. Bäume

Definition, Binärbäume, Implementierung

- **Def.**: Ein **Baum** besteht aus einem ausgezeichneten Knoten r (**Wurzel**, *root*) und k ≥ 0 disjunkten Bäumen T1, ..., Tk
- **Binärbäume: Def.**: Ein **Binärbaum** B = (r, BL, BR) ist entweder leer oder besteht aus einer Wurzel r sowie einem linken und rechten Teilbaum BL bzw. BR
- ```
  typedef struct tr
  { T* data;
    struct tr *left,*right; }
  btree, *btreeptr;
  ```
- Anwendung z.B. für Symboltabelle eines Compilers (enthält Informationen zu Variablen eines Programms), Variablendeklaration → neuer Eintrag in Tabelle mit Suchschlüssel ID, Verwendung → Suchen nach ID, nicht mehr benötigt → Löschen

Baumdurchläufe: Ablauf & Implementierung

- zur Verarbeitung aller Knoten in einem Baum
- L → Durchlauf linker Teilbaum, R → Durchlauf rechter Teilbaum, W → Behandlung der Wurzel
- Pre-, In-, Post- bzgl. Zeitpunkt der Wurzel-Bearbeitung

Rekursive Implementierung (Laufzeit: O(# Knoten)):

<u>Preorder</u>

- WLR
- ```
  void preorder(btreeptr b)
  {
   if (b == NULL) return;
   visit(b->data);
   preorder(b->left);
   preorder(b->right);
  }
  ```

<u>Inorder</u>

- LWR
- ```
  void inorder(btreeptr b)
  {
   if (b == NULL) return;
   inorder(b->left);
   visit(b->data);
   inorder(b->right);
  }
  ```

<u>Postorder</u>

- LRW
- ```
  void postorder(btreeptr b)
  {
   if (b == NULL) return;
   postorder(b->left);
   postorder(b->right);
   visit(b->data);
  }
  ```

Suchbäume

<u>Definition</u>

- **Def**.: Sei B = (r, BL, BR) ein Binärbaum und bezeichne key(n) den Suchschlüssel für jeden Knoten n. B heißt **Suchbaum**, falls gilt:
  1. Für alle $n \in BL: key(n) < key(r)$
  2. Für alle $n \in BR: key(n) > key(r)$
  3. BL und BR sind Suchbäume,

  Voraussetzung: Kein Schlüssel kommt doppelt vor (echt kleiner & echt größer!)
- für schnellere Mengenoperationen gegenüber Listen
- Binärer Suchbaum: Entscheidungsbaum bei der binären Suche in Arrays als explizite Datenstruktur

<u>Operationen & Laufzeiten</u>

- Einfügen eines Elements

```
btreeptr insert(T* item, btreeptr b)
{ btreeptr n;
  if (b == NULL)
      { n = (btreeptr)malloc(sizeof(btree));
        n->data = item;
        n->left = NULL; n->right = NULL;
        return n;
      }
  if (key(item) < key(b->data))
      b->left = insert(item,b->left);
  if (key(item) > key(b->data))
      b->right = insert(item,b->right);
  return b;
}
```

  left- bzw. right-Zeiger werden durch neuen Teilbaum ersetzt

- Suchen eines Elements:

```
T* search(T* item, btreeptr b)
{
 if (b == NULL) return NULL;
 if (key(item) == key(b->data))
      return b->data;
 if (key(item) < key(b->data))
      return search(item,b->left);
 if (key(item) > key(b->data))
      return search(item,b->right);
}
```

  Laufzeit: durch Pfadlänge bestimmt, min. O(log n) (ausgeglichener Baum), max. O(n) (entarteter Baum, wie Liste), average case: O(log n), n = Anzahl Knoten
- Löschen eines Elements: gleiches Prinzip wie Suche, aber Suchbaum muss noch wiederhergestellt werden
- AVL-Bäume (ausgeglichene Suchbäume): O(log n) auch im worst case für Suchen, Einfügen, Löschen

6. Graphen

Definition und Eigenschaften

- **Def.**: Ein **(gerichteter) Graph** G = (V,E) besteht aus einer Menge V von **Knoten** (*vertices*) und einer Menge E von **Kanten** (*edges*) mit $E \subseteq V \times V$
- ungerichtet: Gilt für alle Kanten (u,v) eines Graphen G = (V,E) $(u,v) \in E \Rightarrow (v,u) \in E$, so heißt G **ungerichtet**. **Schreibweise**: $\{u,v\} \in E$
- Pfad: Eine Folge (v1, ..., vn) von Knoten eines Graphen G = (V,E) heißt **Pfad**, falls für alle i = 1...n-1: $\{vi,vi+1\} \in E$
- Zusammenhängend: Ein Graph G = (V,E) heißt **zusammenhängend**, falls zwischen zwei beliebigen Knoten $u,v \in V$ mindestens ein Pfad existiert
- Zyklenfrei: Ein Graph G = (V,E) heißt **zyklenfrei**, falls zwischen zwei beliebigen Knoten $u,v \in V$ höchstens ein Pfad existiert
- Baum: Ein zusammenhängender, zyklenfreier Graph G = (V,E) heißt **Baum**
- gewichtet: Ein Graph G = (V,E,w) mit einer Abbildung $w : E \to \mathbf{R}$ heißt **(kanten)gewichteter Graph**

Dijkstra-Algorithmus

Definitionen & Mengen

- zur Bestimmung der kürzesten Pfade von einem Knoten v_0 zu allen anderen Knoten
- Idee: Menge M von Knoten („Nachbarschaft von v_0“), deren kürzeste Abstände zu v_0 bekannt sind, M immer um den Knoten erweitern, der am nächsten an M liegt
- Bisher bekannter min. Abstand *dist(u,v)* zwischen zwei Knoten kann durch neue Zwischenknoten verkürzt werden
- **Def.**: Ist G = (V,E) ein Graph und $\{u,v\} \in E$, so heißt v **Nachfolger** von u, und u heißt **Vorgänger** von v
- Menge GREEN: Knoten, deren kürzester Abstand zu v0 bekannt ist, YELLOW: Nachfolger der Knoten in GREEN, RED: Kanten, die kürzeste Wege bilden
- dist[v]: min. Abstand von v zu v0
- cost(v,w): Kantengewicht der Kante {v,w}
- graph und set mit Operationen als Datentypen vorhanden

Ablauf

Implementierung

Pseudo-Code:

```
void shortest_path(graph G = (V = {v0,...,vn},E))
{ set GREEN = {}, YELLOW = {v0}, RED = {}; dist[v0] = 0;
  while (YELLOW != {}) /* solange noch unbearbeitete Knoten */
      { v = MinDist(YELLOW);  /* v mit min. Dist-Wert → YELLOW */
        Insert(v,GREEN);
        Delete(v,YELLOW);

        for (w Î Succ(v)) /* für alle Nachfolger w von v */

            { if (!(w Î YELLOW È GREEN)) /* neuer Knoten erreicht → neuer Weg */
                  { Insert({v,w},RED);
                    Insert(w,YELLOW);
                    dist[w] = dist[v] + cost(v,w);
                  }
            else if (w Î YELLOW)  /* w erneut erreicht */
                  { if (dist[v] + cost(v,w) < dist[w])
```

```
                        { Insert({v,w},RED);
                          e = PreviousEdge(w); /* vorher rote Kante zu w */
                          Delete(e,RED);
                          dist[w] = dist[v] + cost(v,w);
                        }
                    }
                }
            }
}
```

Laufzeitanalyse: für jeden neuen „grünen“ Knoten: MinDist → O(V), Verarbeitung der O(V) Nachfolger (in beiden if-Fällen O(1) Schritte) → O(V), Gesamt: O(V^2)
Mit besserer Implementierung: O(E*log V) (besser wenn E<<V^2)

Repräsentationen: Eigenschaften & Implementierung von ...

<u>… Adjazenzmatrix</u>

- **Def.**: Ist G = (V,E) ein Graph mit Knotenmenge $V = \{v_1, ..., v_n\}$, so heißt die (n × n)-Matrix $A = (a_{ij})$ mit
  $a_{ij} = 1$, falls $\{v_i, v_j\} \in E$ (v_i & v_j sind durch eine Kante verbunden)
  $a_{ij} = 0$, sonst
  Adjazenzmatrix zu G (für ungewichtete Graphen)
- zweidimensionales Array, Knotenmenge implizit in Anzahl der Zeilen / Spalten enthalten
- bei kantengewichteten Graphen: Kantengewicht statt 0 / 1, spezieller Wert für „nicht verbunden“
- Vorteil: Adjazenztest in O(1)
- Nachteil: Platzbedarf O(V^2)
- Implementierung:

```
#define N 100

int adj_matrix[N][N];

int adjacent (int i, int j)
{ return adj_matrix[i][j];)
}
```

<u>… Adjazenzlisten</u>

- für jeden Knoten Liste seiner Nachbarn speichern → Kanten werden explizit gespeichert
- Array von verketteten Listen
- Vorteil: Platzbedarf O(E+V), gut vor allem bei wenigen Kanten
- Nachteil: Adjazenztest im worst case in O(V)
- Implementierung:

```
#define N 100
List adj_list[N];

int adjacent(int i,int j)
{
 List L;
 L = adj_list[i];
 return IsIn(j,L);
}
```

Graphdurchläufe: Implementierung & Laufzeiten von

- um alle Knoten eines Graphen zu verarbeiten, z.B. Test ob Graph zusammenhängend
- Bsp: garbage collection für Heap-Speicher → Test auf nicht mehr erreichbare Speicherblöcke

<u>DFS</u>

- Depth-First-Search (Tiefensuche) → vom Startknoten aus möglichst lange Pfade verfolgen
- Implementierung (Pseudo-Code):

```
int visited[N]; /* implizit mit 0 initialisiert */
void dfs(int v) /* DFS beginnend bei v */
{ int w;
  visited[v] = 1; /* markiere v als besucht */
  process(v); /* Verarbeitung von v */
  for ({v,w} Î E) /* für alle Nachbarn w */
      { /* rekursiver Aufruf von dfs: */
        if (!visited[w]) dfs(w);
      }
}

void main()
{ int v;
  for (v = 0; v < N; v++)
      if (!visited[v]) dfs(v);
}
```

- bei zusammenhängendem Graphen nur ein Aufruf von dfs nötig, sonst von allen Knoten aus
- Laufzeit: Aufruf von dfs() für jeden Knoten: O(V), einmalige Verarbeitung aller Kanten: O(E) → Gesamt: O(V+E)

<u>BFS</u>

- Breadth-First-Search (Breitensuche) → vom Startknoten aus ebenenweise alle Knoten besuchen
- Queue, um Knoten, die gerade besucht werden, als Ausgangsknoten für nächste Ebene zu speichern
- Implementierung (Pseudo-Code):

```
int visited[N];
void bfs(int v) /* BFS ab Knoten v */
{ int w; Queue Q;
  visited[v] = 1;
  process(v);
  Put(v,Q);
  while (!IsEmpty(Q))
      { v = Get(Q);
        for ({v,w} Î E)
            { if (!visited[w])
                  { visited[w] = 1;
                    process(w);
                    Put(w,Q);
                  }
            }
      }
}

void main()
{ int v;
  for (v = 0; v < N; v++)
```

```
        if (!visited[v]) bfs(v);
    }
```

- Laufzeit: Aufruf von bfs() für alle Knoten: O(V), einmalige Verarbeitung aller Kanten: O(E) → Gesamt: O(V+E)

Kruskal-Algorithmus

- Spannbaum: Ein **Spannbaum** in einem ungerichteten, zusammenhängenden und kantengewichteten Graphen G = (V,E,w) ist ein zyklenfreier, zusammenhängender Teilgraph G´= (V´,E´,w) mit V´ = V und E´ ⊆ E
- Minimaler Spannbaum: Ein **minimaler Spannbaum** in einem ungerichteten, zusammenhängenden und kantengewichteten Graphen G = (V,E,w) ist ein Spannbaum mit minimaler Kantengewichtssumme unter allen Spannbäumen zu G (nicht eindeutig!)

<u>Ablauf</u>

- aufsteigend nach Gewicht sortierte Liste der Kanten muss vorliegen
- Beginn mit leerem Spannbaum
  → Auswahl der Kante mit kleinstem Gewicht → wenn Hinzunahme Zyklus verursachen würde, verwerfen, sonst in Spannbaum aufnehmen → Fortsetzen bis Spannbaum n-1 Kanten enthält

<u>Implementierung (Pseudo-Code)</u>

```
Graph MinSpanningTree(Graph G = (V,E))
{ Graph T; List L; Edge e;
  T = {};
  L = Sort(E); /* sortierte Kantenliste */
  while (#Kanten in T < #Knoten in G - 1 && !IsEmpty(L))
      { e = FirstElem(L); /* nächst-billigste Kante */
        Delete(e,L);
        if (!Cycle(e,T)) /* kein Zyklus in T È {e} ? */
            { T = T È {e};
            }
      }
  if (#Kanten in T < #Knoten in G - 1)
        { /* keine verfügbaren Kanten mehr, d.h. G ist nicht zusammenhängend */
          printf(„Fehler - kein Spannbaum möglich!“);
        }
  return T;
}
```

<u>Laufzeit</u>

- dominiert durch Sortierung der Kanten → O(E*log E)
- while-Schleife: max. bis Kantenliste leer ist → O(E), Entnahme & Löschen von e: O(1), Zyklus-Test O(1)
- Gesamt: O(E*log E)

Techniken zum Algorithmenentwurf

Rekursion & Iteration

- Rekursion: Berechnung durch erneuten Aufruf der Funktion, zusätzlicher Speicherbedarf und Aufwand wegen Laufzeit-Stack
- Iteration: Berechnung durch sich wiederholende Schritte, meist bessere Laufzeit

- Bsp Fibonacci-Folge:
  Rekursive Variante: fib(n-2) wird unnötigerweise immer doppelt berechnet → Laufzeit: O(2^n)
  Iterative Variante: Ausnutzen der schon berechneten Werte → Laufzeit: O(n)