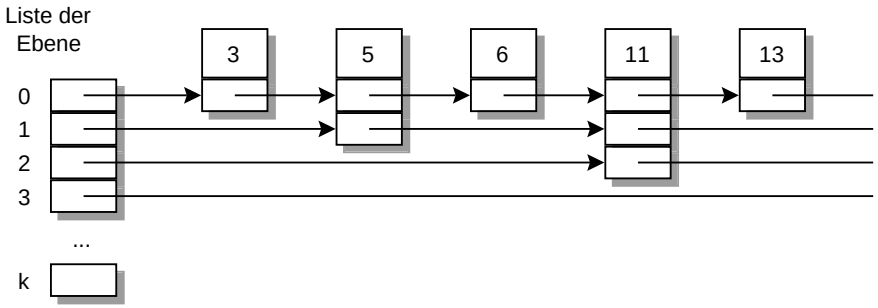
Institut 186 für Computergraphik	**Algorithmen und Datenstrukturen 1**				VO-Prüfung 18. Juni 1998 Gruppe A	
Kennz.†	**Matrikelnr.**	**Beil.**‡	1 (30)	2 (15)	3 (25)	4 (30)
Nachname, Vorname						Σ (100)

† Geben Sie jene Kennzahl an, auf die das Zeugnis ausgestellt werden soll.
‡ Geben Sie an, wieviele Zusatzbl. Sie abgeben (**jedes mit Name & Matr.-Nr. beschriftet!**).

(30) Aufgabe 1 – Skip Lists

Skip Lists sind eine Datenstruktur, die zur schnellen Suche von Datensätzen nach einem Schlüssel dienen. Sie basieren auf sortierten Listen. Die Idee liegt darin, daß jeder Datensatz in einer nach dem Schlüssel sortieren Liste enthalten ist. Dies ist die Liste der Ebene 0. Jeder zweite Datensatz ist in einer weiteren sortierten Liste enthalten, der Liste der Ebene 1. Jeder vierte Datensatz ist in einer weiteren sortierten Liste der Ebene 2 enthalten usw. bis zur Liste der Ebene k, die leer ist. Graphisch kann man sich den Sachverhalt so vorstellen:



a) Entwerfen Sie Datenstrukturen `SkipNode` und `SkipList` für Datensätze und Skiplists von `INTEGER` Zahlen. Gehen Sie davon aus, daß maximal 2^{24} Datensätze in ihren Skiplists gespeichert werden müssen. Sie brauchen dabei nicht darauf zu achten, die Pointer effizient abzuspeichern: sehen Sie in ihren Datenstrukturen genügend Pointer für den schlimmsten Fall vor.

b) Schreiben Sie eine *effiziente* Funktion

```
FUNCTION search(list : SkipList; key : INTEGER) : SkipNodePtr;
```

die nach dem Datensatz mit dem Schlüssel `key` in der Skiplist `list` sucht, und einen Pointer auf ihn zurückliefert, wenn er gefunden wurde. Sollte der Datensatz nicht in der Liste sein, so soll `Nil` zurückgegeben werden.

c) Wieviele der Pointer in jedem Datensatz werden durchschnittlich tatsächlich gebraucht? Oder mit anderen Worten: Wenn nun in jedem Datensatz die Pointer der Skipliste *effizient* gespeichert werden, wieviele Pointer sind durchschnittlich pro Datensatz notwendig? (*Geben Sie eine Begründung für Ihr Ergebnis an!*)

(15) Aufgabe 2 – Radix Exchange Sort

Gegeben ist die folgende, unsortierte Zahlenfolge:

```
40 61 15 30 18 49 24 2 21 35 58 53 6 33 43 38
```

Sortieren Sie diese Zahlenfolge *absteigend* mit **Radix Exchange Sort**. Verwenden Sie den Algorithmus so, wie er im Skriptum steht. **Geben Sie jeden Zwischenschritt an**, in der Tabelle ist ausreichend Platz. Achten Sie darauf, daß **in jeder Zeile nur Teilfolgen derselben Rekursionstiefe** stehen.

Kennzeichnen Sie deutlich die Grenzen der Teilfolgen!

Als Hilfe enthält die erste Zeile der Tabelle die Indices der Elemente.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
40	**61**	**15**	**30**	**18**	**49**	**24**	**2**	**21**	**35**	**58**	**53**	**6**	**33**	**43**	**38**

Hilfe:

0	000000	16	010000	32	100000	48	110000
1	000001	17	010001	33	100001	49	110001
2	000010	18	010010	34	100010	50	110010
3	000011	19	010011	35	100011	51	110011
4	000100	20	010100	36	100100	52	110100
5	000101	21	010101	37	100101	53	110101
6	000110	22	010110	38	100110	54	110110
7	000111	23	010111	39	100111	55	110111
8	001000	24	011000	40	101000	56	111000
9	001001	25	011001	41	101001	57	111001
10	001010	26	011010	42	101010	58	111010
11	001011	27	011011	43	101011	59	111011
12	001100	28	011100	44	101100	60	111100
13	001101	29	011101	45	101101	61	111101
14	001110	30	011110	46	101110	62	111110
15	001111	31	011111	47	101111	63	111111

(25) Aufgabe 3 – Zur Fußballweltmeisterschaft

Ein Programmierer erhält die Aufgabe, Programme zur statistischen Auswertung einzelner Spiele bei der Fußballweltmeisterschaft zu schreiben. Hierzu wurden die Positionen der Spieler in jedem Spiel in regelmäßigen Abständen in Quadtrees abgespeichert. Es soll nun herausgefunden werden wieviele Spieler jeder Mannschaft sich zu den aufgezeichneten Zeitpunkten jeweils in den Strafräumen und in den beiden Spielhälften befunden haben. Die Datenstruktur für den Quadtree sieht folgendermaßen aus:

```
TYPE
    TeamTyp = (Eins, Zwei);
    KnotenPtr = ^Knoten;
    Knoten = RECORD
               x, y: REAL;   { Spielerposition }
               team: TeamTyp { welche Mannschaft }
               next: ARRAY[1..4] OF KnotenPtr;
             END;
```

Jeder Quadtree-Knoten enthält die Information über einen Spieler, sowie 4 Zeiger auf die vier von dieser Spielerposition definierten Quadranten.

Schreiben Sie nun eine *effiziente* Prozedur:

```
FUNCTION countPlayers(spielfeld : KnotenPtr;
                      xmin,xmax,ymin,ymax : REAL
                      VAR anzahleins, anzahlzwei : INTEGER);
```

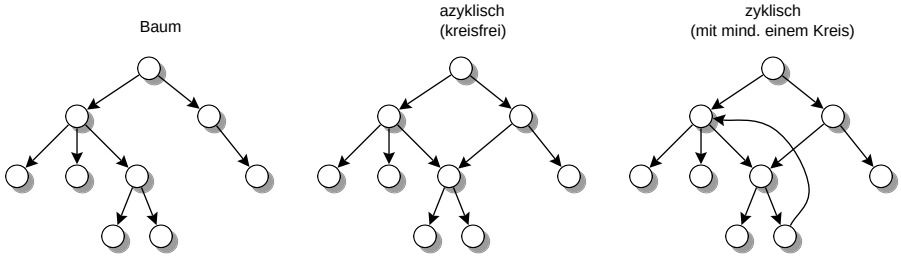
die die Anzahl der Spieler jeder Mannschaft berechnet, die sich im übergebenen rechteckigen Bereich aufhält, und diese in `anzahleins` bzw. `anzahlzwei` zurückliefert.

(30) Aufgabe 4 – Graphen

Folgende Definition für Adjazenzlisten von gerichteten Graphen sei gegeben:

```
TYPE
    KnotenPtr = ^Knoten;
    Knoten = RECORD
               nummer : INTEGER;
               next : KnotenPtr;
             END;
    GraphTyp = ARRAY[1..K] OF KnotenPtr;
```

Man kann folgende drei Arten von gerichteten Graphen unterscheiden: Bäume, azyklische Graphen und zyklische Graphen. Hier ein Beispiel:



Diese drei Arten von Graphen seien in folgendem Enumerationstyp definiert:

```
GraphArt = (Baum, Azyklisch, Zyklisch);
```

Schreiben Sie nun eine *effiziente, rekursive* Funktion

```
FUNCTION welcherGraph(graph : GraphTyp;
                      wurzelNummer : INTEGER) : GraphArt;
```

die, wenn man einen Graphen (`graph`) und einen Wurzelknoten (`wurzelNummer`) übergibt, berechnet, um welche Art von Graph es sich handelt.

Geben Sie auch alle Hilfsfelder an die Sie benötigen, und mit welchen Werten sie initialisiert werden müssen! *Hinweis:* ein einziges Hilfsfeld vom Typ `BOOLEAN` reicht nicht aus, um alle drei Arten voneinander unterscheiden zu können!