<u>Musterlösung:</u> AlgoDat (Magister) 3. Serie

1.) 2 Pkt. Liste L

L = | E_1 | $Rest_L$ |

$Rest_L$ = | E_2 | $Rest_{Rest}$ |

$Rest_{Rest} = \varnothing$

so L = | E_1 | E_2 | $\varnothing$ | L enthält zwei Elemente

2.) 3 Pkt. Liste L=(1,6,0,7,1)

(a) Länge & Anzahl der Positionen:
Länge(L) = 5
#Positionen = 5

(b) Angabe der Teillisten:
(Teilliste ist eine zusammenhängende Teilsequenz)

1. (1,6,0,7,1)
2. (1,6,0,7)
3. (1,6,0)
4. (1,6)
5. (1)
6. ()
7. (6,0,7,1)
8. (6,0,7)
9. (6,0)
10. (6)
11. (0,7,1)
12. (0,7)
13. (0)
14. (7,1)
15. (7)

16. (1) die Teilliste (1) existiert zweimal

(c) Anzahl der Teilsequenzen:
(Entfernung von 0-n Elementen einer Liste, Ordnung der Elemente der Liste bleibt bestehen)
Teillisten sind demnach auch Teilsequenzen.
<u>Zusätzlich</u> erhält man:

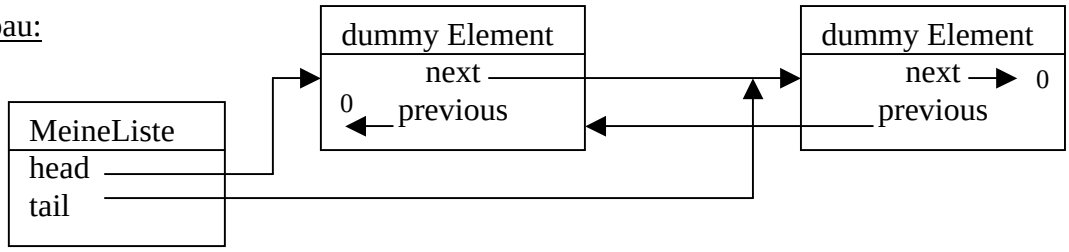
0 El. weglassen	1El. weglassen	2 El. weglassen	3 El. weglassen	4 El. weglassen	5 El. weglassen
-	1671 1071 1601	167 161 171 107 101 671 601	10 67 61 17 11 01	-	-
= 0	= 3	= 7	= 6	= 0	= 0

Gesamt Sequenzen = 16 Teillisten + 16 zusätzl. Teilsequenzen = 32
(unter der Beachtung, dass die Teilliste *(1)* zweimal auftritt)

Dies ist auch plausibel, da die Anzahl der Elemente einer Powermenge von M mit n Elementen gleich 2^n Elemente besitzt. Die Liste L kann als Menge M dargestellt werden, indem man einen Index generiert und die Menge M geordnet hält (keine Mengeneigenschaft!). Z.B. M={0,1,2,3,4}, so #Elemente(Pow(M)) = 32 = #Teilsequenzen

3.) 10 Pkt. (+ 1 Zusatzpkt. für Syntax) Listenprogrammierung

Aufbau:



(a) Initialisierung

```
MeineListe = (Liste *) malloc( sizeof( Liste ) );
MeineListe->head = NULL;
MeineListe->tail = NULL;
MeineListe->size = 0;
// Initialisierung einer Liste mit dummy Elementen
element = (ListElmt *) malloc(sizeof(ListElmt));
element->previous = NULL;
element->Inhalt = "1";
MeineListe->head = element;
element = (ListElmt *) malloc(sizeof(ListElmt));
element->previous = MeineListe->head;
element->next = NULL;
element->Inhalt = "2";
MeineListe->head->next = element;
MeineListe->tail = element;
```

(b) Einfügen von new_element am Ende

```
// Initialisieren
new_element = (ListElmt *) malloc(sizeof(ListElmt));
new_element->Inhalt = "n";
// Pointer auf das Ende setzen
element = MeineListe->tail;
// Pointer umhängen und somit new_element einfügen
element->previous->next      = new_element;
new_element->previous        = element->previous;
new_element->next    = element;
element->previous    = new_element;
// size ändern
MeineListe->size += 1;
```

(c) Einfügen von new_element am Anfang

```
// vorher new_Element initialisieren wie bei (b)
// Pointer auf den Anfang setzen
element = MeineListe->head;
// Pointer umhängen und somit new_element einfügen
new_element->next = element->next;
new_element->previous = element;
new_element->next->previous = new_element;
element->next = new_element;
// size ändern
MeineListe->size += 1;
```

(d) Einfügen von new_element hinter *element (bei einfacher Verkettung)

```
// vorher new_Element initialisieren wie unter (b)
new_element->next = element->next;
element->next = new_element;
// size ändern
MeineListe->size += 1;
```

(e) Einfügen von new_element hinter *element (bei doppelter Verkettung)

```
// vorher new_Element initialisieren wie unter (b)
// Pointer auf den Anfang setzen
new_element->next = element->next;
new_element->previous = element;
element->next->previous = new_element;
element->next = new_element;
// size ändern
MeineListe->size += 1;
```

4.) 6 Pkt. Stackanwendung

(a) ab+cd*+e* **Stack S**

push(S,a)	a				
push(S,b)	b	a			
push(pop(S) + pop(S))	(b+a)				
push(S,c)	c	(b+a)			
push(S,d)	d	c	(b+a)		
push(pop(S) * pop(S))	(d*c)	(b+a)			
push(pop(S) + pop(S))	((d*c)+(b+a))				
push(S,e)	e	((d*c)+(b+a))			
push(pop(S) * pop(S))	(e*((d*c)+(b+a)))				

(b) abcd+++ **Stack S**

push(S,a)	a				
push(S,b)	b	a			
push(S,c)	c	b	c		
push(S,d)	d	c	b	a	
push(pop(S) + pop(S))	(d+c)	b	a		
push(pop(S) + pop(S))	((d+c)+b)	a			
push(pop(S) + pop(S))	(((d+c)+b)+a)				

(Die Klammerung spielt hier keine Rolle.)

(c) ab+c+d+e+ **Stack S**

push(S,a)	a				
push(S,b)	b	a			
push(pop(S) + pop(S))	(b+a)				
push(S,c)	c	(b+a)			
push(pop(S) + pop(S))	(c+(b+a)				
push(S,d)	d	(c+(b+a)			
push(pop(S) + pop(S))	(d+(c+(b+a)))				
push(S,e)	e	(d+(c+(b+a)))			
push(pop(S) + pop(S))	(e+(d+(c+(b+a)))				

(Die Klammerung spielt hier keine Rolle.)

5.) 2 Pkt. Stackanwendung

Operationen	**Stack S**				
push(S,a)	a				
push(S,b)	b	a			
pop(S)	a				
push(S,c)	c	a			
push(S,d)	d	c	a		
pop(S)	c	a			
push(S,e)	e	c	a		
pop(S)	c	a			
pop(S)	**a**				

Der Stack S entält nach Abarbeitung der Operationen das Zeichen a.

6.) 2 Pkt. Postfixschreibweise

$$3*(7+4)/(2+11)-3 \equiv 74+3*2(11)+/3-$$

Bei dieser Darstellung ist stets die vorderste Zahl, auch der erste Operand für die anstehende Operation. (d.h. $ab- \equiv a-b$)