アルゴリズムとデータ構造

- 第2回講義トピック：基本データ構造
  - なぜデータ構造か
  - 配列
  - 連結リスト

Algorithms and Data Structures

- Topic of lecture No. 2: basic data structures
  - why data structure?
  - array
  - linked list

なぜデータ構造か

- データ構造とは情報を保存し、組織するためのメカニズム
- 例えば、
  - 配列は同じタイプのデータを連続したスペースに配置し、インデックスによってランダムにアクセスすることができる。
  - 構造体はいろんなタイプのデータを1つのまとまりとして扱うことができる。
  - リストは同じタイプのデータを扱うが、連続したスペースに置く必要がなく、ランダムではなく、シーケンシャルアクセスにのみ対応する
- データ構造が異なれば、必要となる記憶領域が変わるし、アルゴリズムの効率も違ってくる。
- データ構造とアルゴリズムの選択は互いに深く結びついており、この選択を上手に行うことにより、実行時間と記憶領域の節約が可能となる。
- よいデータ構造を選択することによって、効率のよい、明快なアルゴリズムを実現できる。

Why data structure?

- Data structure is the way to organize data.
  Different data structures can be used for different algorithms.
  Choosing a better or more suitable data structure can make the algorithm simpler and faster.
- Examples:
  - An array contains same type of data in a continuous memory block with fixed size.
    The elements in an array can be accessed randomly using indexes.
  - A record (structure in C) contains a collection of fields with different data types.
  - A (linked) list contains a sequence of same data type. The elements are not necessary to be located in a continuous memory block. They can only be accessed in a sequential manner.

この授業で扱う基本的なデータ構造

- 配列
- リスト
- スタック
- キュー
- 木
- グラフ
- …

Main data structures in this course

- array
- list
- stack
- queue
- tree
- graph
- …

配列(array)

- 配列は、同じタイプの要素となるデータを一定個数メモリ上の連続した空間に並べたもので、各要素は添字かポインタを用いて参照される。
  - 配列aの第i要素はa[i]か*(a+i)で表す。
  - 配列はデータ構造の最も基本的なもので、
  - コンピュータの記憶方式に直接対応する

Array

- Data structure for a block of fixed size contains same type of elements which can be randomly access by indexes (or by points in C language).
  - the number i element: a[i] or *(a+i)
  - a simple but basic data structure
  - corresponds directly to memory arrangement of computers

配列の例

- 多項式の係数を配列を用いて表す
  - $x^{14} + 3x^5 + 12$ の配列による表現

```
int p[15];  //領域の確保
for (i=0; i<15; i++) p[i]=0; //全体の初期化
p[0]=12;p[5]=3;p[14]=1; //データのセット
```

Examples using arrays

- Used for the coefficients of polynomials
  - $x^{14} + 3x^5 + 12$

```
int p[15];  // allocation for all possible values
for (i=0; i<15; i++) p[i]=0; // initialize to 0
p[0]=12;p[5]=3;p[14]=1; // set the coefficients
```

エラトステネスの篩

- 指定した整数以下の素数をすべて出力するアルゴリズム
- 古代ギリシアの科学者エラトステネスが紀元前3世紀ごろに考えたアルゴリズム
  - ブール値配列a[i]で素数かどうかを表す
    0：素数でない；　1：素数である
  - 篩とはある値iが素数でないと判定された場合は、iの任意の倍数もすべて素数ではないという性質を利用するものである。

Sieve of Eratosthenes

- Ancient algorithm for finding all prime numbers up to a given limit number by ancient Greek mathematician Eratosthenes.
  - use a boolean type array a[i]
    0: not prime number; 1: prime number
  - The sieve marks the multiples of each prime as composite (i.e., not prime) starting from 2.

エラトステネスの篩のプログラム例

```
#include <stdio.h>
#define N 1000
void main()
{
  int i, j, a[N+1];  // データ領域確保
  a[1] = 0;
  for(i = 2; i <= N; i++)
    a[i] = 1; // 全ての数に1をセット、篩の用意をする
  for(i=2; i<=N/2; i++)
    for(j=2;j<=N/i;j++)
      a[i*j] = 0;  //素数でないもの(i*j)を篩い落とす
  for(i = 1; i <= N; i++)
    if (a[i]) printf("%4d", i);
  printf("¥n");
}
```

注）このプログラムでは配列aのインデックスは 1 から使う

A simple programming

```
#include <stdio.h>
#define N 1000
void main()
{
  int i, j, a[N+1];  // data array
  a[1] = 0;
  for(i = 2; i <= N; i++)
    a[i] = 1; // preset to 1 (prime number)
  for(i=2; i<=N/2; i++)
    for(j=2;j<=N/i;j++)
      a[i*j] = 0;  // sifts not prime numbers
  for(i = 1; i <= N; i++)
    if (a[i]) printf("%4d", i);
  printf("¥n");
}
Note: this program uses array from index 1
```

すこし効率を改善した解答

```
#include <stdio.h>
#define N 1000
void main()
{
  int i, j, a[N+1]; // データ領域確保
  a[1] = 0;
  for(i = 2; i <= N; i++)
    a[i] = 1; // 全ての数に1をセット、篩の用意をする
  for(i=2; i<=N/2; i++)
    if (a[i]) //iが素数の場合だけ処理が必要
      for(j=2;j<=N/i;j++)
        a[i*j] = 0; //素数でないもの(i*j)を篩い落とす
  for(i = 1; i <= N; i++)
    if (a[i]) printf("%4d", i);
  printf("¥n");
}
```

Slightly improved programming

```
#include <stdio.h>
#define N 1000
void main()
{
  int i, j, a[N+1];  // data array
  a[1] = 0;
  for(i = 2; i <= N; i++)
    a[i] = 1; // preset to 1 (prime number)
  for(i=2; i<=N/2; i++)
    if (a[i]) // necessary only for primes
      for(j=2;j<=N/i;j++)
        a[i*j] = 0;  // sifts not prime numbers
  for(i = 1; i <= N; i++)
    if (a[i]) printf("%4d", i);
  printf("¥n");
}
```

Note: this program uses array from index 1

最も効率のいいエラトステネスの篩

```
Input: an integer n > 1.
Let A be an array of Boolean values,
  indexed by integers 2 to n,
  initially all set to true.

for i = 2, 3, 4, ..., not exceeding √n:
   if A[i] is true:
     for j = i*i, i*(i+1), i*(i+2), i*(i+3),
     ..., not exceeding n:
       A[j] := false.

Output: all i such that A[i] is true.
```

Note: i*(i-1), i*(i-2), … はiの値がi-1, i-2, …の時にすでに篩い落とされている。

Use indexed for, if loop without endfor or endif
Use := as assignment

The most efficient algorithm

```
Input: an integer n > 1.
Let A be an array of Boolean values,
  indexed by integers 2 to n,
  initially all set to true.

for i = 2, 3, 4, ..., not exceeding √n:
   if A[i] is true:
     for j = i*i, i*(i+1), i*(i+2), i*(i+3),
     ..., not exceeding n:
       A[j] := false.

Output: all i such that A[i] is true.

Note: i*(i-1), i*(i-2), … are already sifted in
the loop when i is i-1, i-2, …
```

Use indexed for, if loop without endfor or endif
Use := as assignment

連結リスト(linked list)

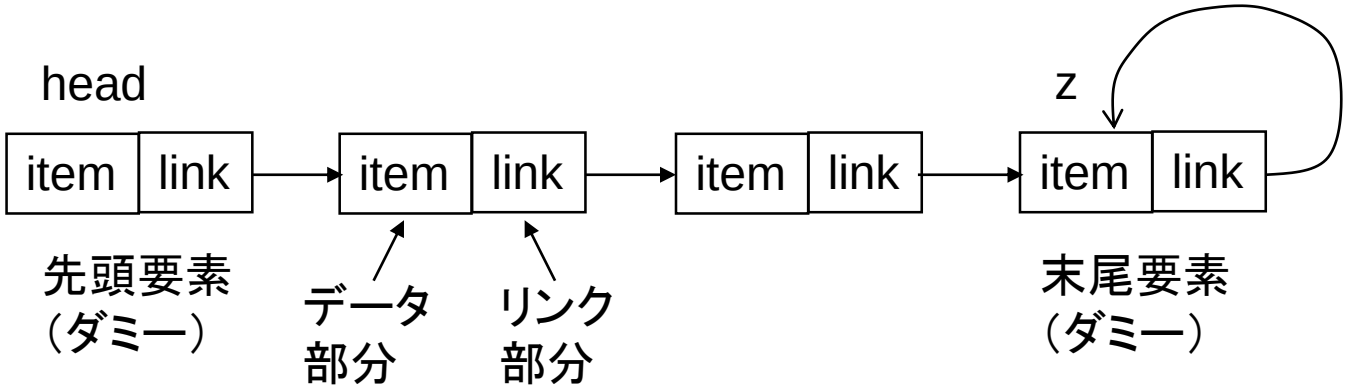
- 連結リストは、リンクによるリスト、また単にリストとも呼ぶ。
- リストは配列と同じように同じタイプのデータ要素を持つ
- リストは、要素となるデータをある線形順序に従ってリンクによってつながれたデータ構造である。
- リストの利点（配列と比較した場合）
  - 各要素となるデータはメモリ上の連続したスペースに配置する必要がなく、任意の位置に置くことができる。
  - 実行中に大きくしたり小さくしたりすることができる。
  - 項目の並べ換えはリンクだけを変えればよいので、速くできる。
- 欠点：項目の参照するは逐次に行うので、効率は相対的に落ちることがある。

Linked list

- Same as arrays, (linked) lists contain same types of data elements.
- The only way to access elements of a list is to follow the link of each element from a know element or its head in a sequential manner.
- Advantages compared with arrays
  - Elements are not necessarily to be located in a same block of memory.
  - Easy to increase or decrease the size by inserting or deleting elements.
  - By changing the links, it can be easy and fast to change the order of the elements.
- Disadvantage: sequential access to each element is slow compared with random access arrays.

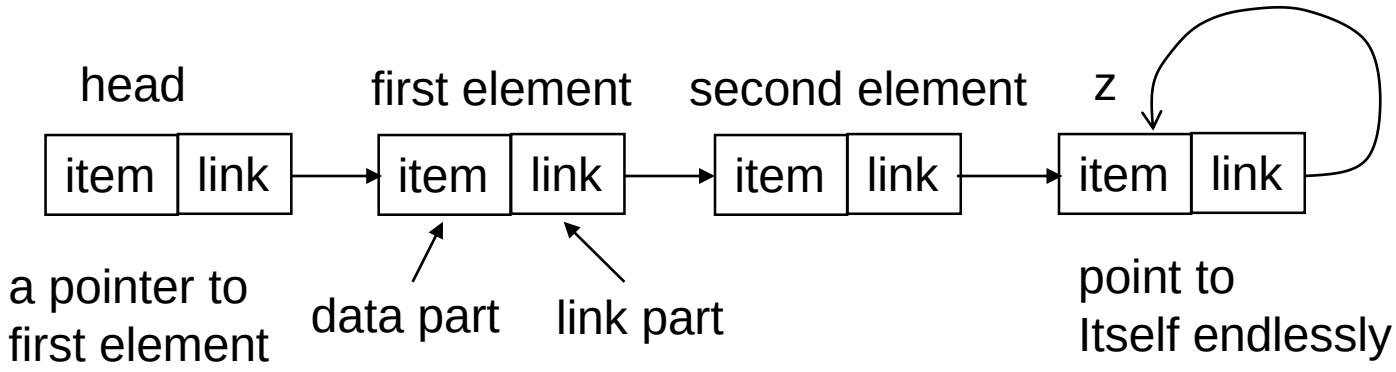
リストの表現

- リストの各要素となるノード(node)はデータの項目(item)と、次のノードの位置を指すリンク(link)からなる。
- 先頭(head)と末尾(tail)ノードはリストそれぞれリストの最初と最後の要素を指す。これらのノードのデータ部分はダミーである。
- ダミーノードzはリストの終了に使われる。zノードがあれば、リンクの有効かどうかを判断する必要がなくなり、処理が速くなる。



Implementation of list

- Each element of a list is constructed by a data type of node structure which contains data part (item) and link part (next).
- The link part is a pointer to the next data element in the list.
- For access to a linked list, we need a special node ‘head’ with dummy data part to point the first data element (if exists). To insert elements to the end of lists, we also need a ‘tail’ node to point the last element.
- For convenient, we usually use a dummy node ‘z’ for the end of list. If there is no element, the head node points to ‘z’ directly. The ‘z’ node points to itself endlessly. By this dummy node, we can always access to the next link without check its availability.



C言語によるリストの表現

- ノードの定義

```
typedef struct node *link; //ノートへのポインタ
struct node {
    ItemType item; //データ部分、リストの構成による
    link next; // あるいは struct node *next;
};
```

- ノードの記憶領域の割付け

```
link x; //ノードへのポインタ
//ノードの領域を確保し、ポインタをセットする
x = malloc(sizeof *x);
```

Implementation by C language

- Data type of node

```
typedef struct node *link; // poiter to a node
struct node {
    ItemType item; // data part of element
    link next; // pointer to the next element
};
```

- Memory allocation of node

```
link x; // a pointer with node type
// allocate memory area to the pointer
x = malloc(sizeof *x);
```

リストの例:

- 多項式 $x^{14} + 3x^5 + 12$のリストによる表現

```
typedef struct node *link;
struct node {
   int coef; int exp; //データ項目部分
   link next; //リンク部分
};
link p, q, r;
p = malloc(sizeof *p); p->coef = 1;  p->exp = 14;
q = malloc(sizeof *q); q->coef = 3;  q->exp = 5;
r = malloc(sizeof *r); r->coef = 12; r->exp = 0;
/* リンクによるリストの連結 */
p->next = q; q->next = r; r->next = NULL;
```

Linked list for polynomials

- For example: $x^{14} + 3x^5 + 12$

```
typedef struct node *link;
struct node {
   int coef; int exp; // data part
   link next; // link part
};
link p, q, r;
p = malloc(sizeof *p); p->coef = 1;  p->exp = 14;
q = malloc(sizeof *q); q->coef = 3;  q->exp = 5;
r = malloc(sizeof *r); r->coef = 12; r->exp = 0;
/* a linked list */
p->next = q; q->next = r; r->next = NULL;
```

Note: It is not for a general implementation.

リストの要素へのアクセス

- すでに存在するheadで始まるリストに対して

```
link p; // ノードへのポインタの定義
p=head->next; //pは最初の要素を指す
p=p->next; //pは2番目の要素を指す
p=p->next; //pは3番目の要素を指す
 //戻ることはできないので、
 //また2番目の要素にアクセスしたい場合は先頭から
p=head->next->next; //pは2番目の要素を指す
```

- データ項目の参照

```
(*p).coef; (*p).exp;
p->coef; p->exp;
```

To access elements of a linked list

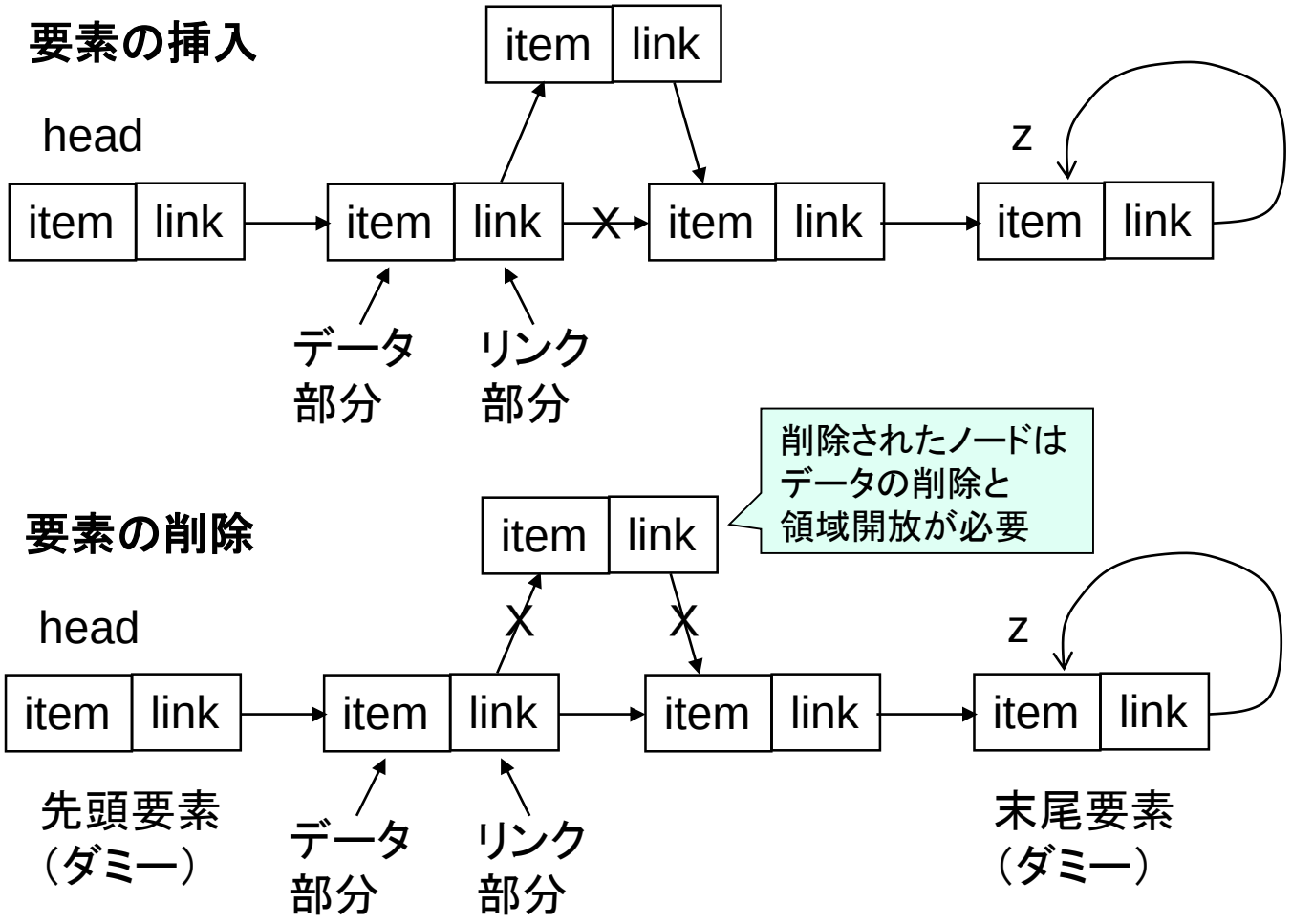
- From ‘head’ pointer

```
link p; // a pointer to node
p=head->next; // p points to the first element
p=p->next; //p points to the second element
p=p->next; //p points to the third element
// For singly linked list,
// there is no pointer the previous element.
// Once passed an element,
// the only way is to go from the head again.
p=head->next->next; //p points to the second element
```

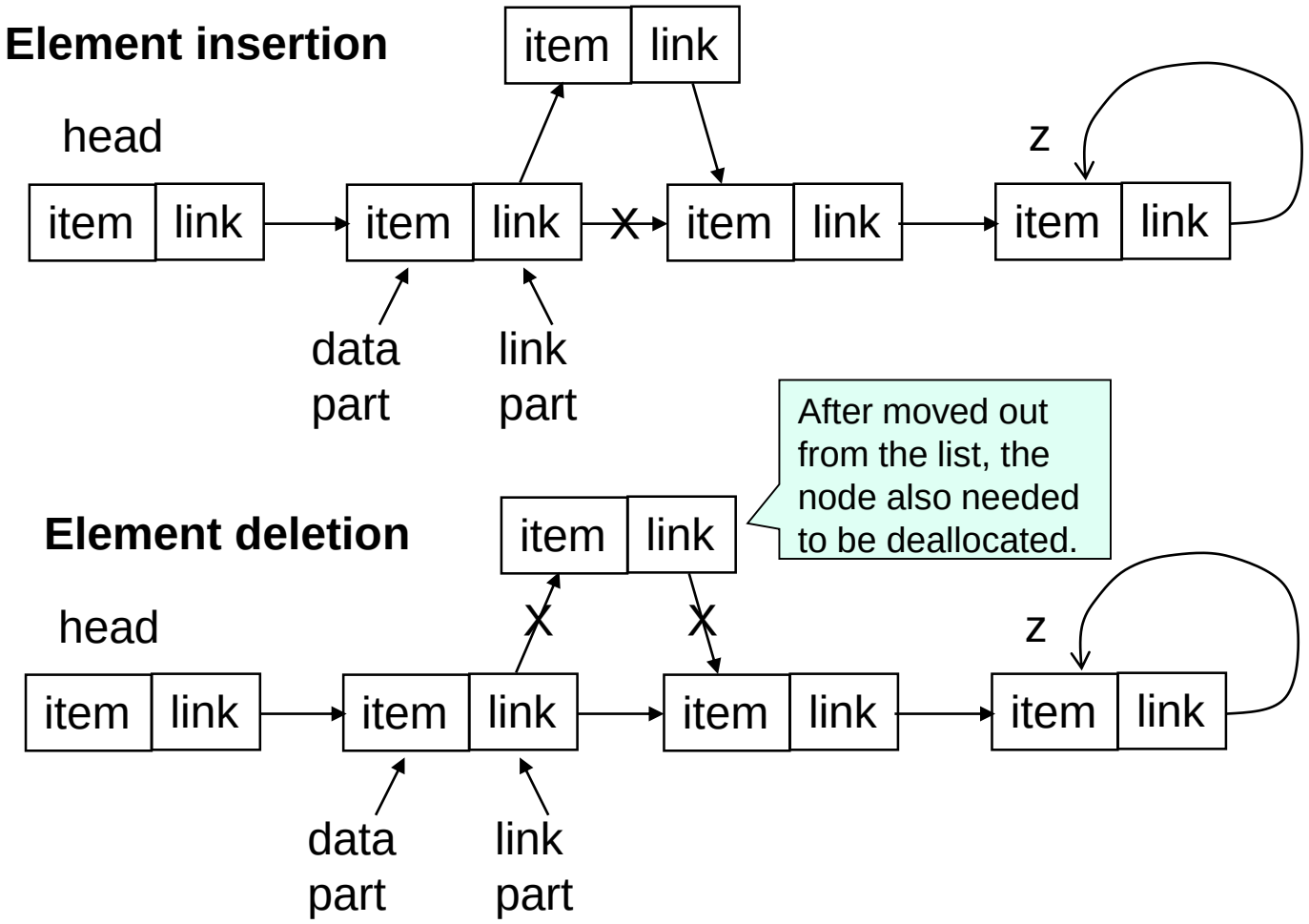
- To refer to data part

```
(*p).coef; (*p).exp;
p->coef; p->exp;
```

リストに対する要素の挿入と削除



Element insertion and deletion of a linked list



初期化、削除、挿入のCプログラムによる実装

```
typedef struct node *link;
struct node { int key; link next; };
link head, z, t;
void listinitialize() {
   head=malloc(sizeof *head); z=malloc(sizeof
*z);
   head->next = z; z->next = z;
}
/* 削除された要素の領域解放はここでは触れない */
void deletenext(link t) { t->next = t->next-
>next; }
link insertafter(int v, link t) {
   link x;
   x = malloc(sizeof *x);
   x->key = v; x->next = t->next; t->next = x;
   return x;
}
```

Implementation in C

```
typedef struct node *link;
struct node { int key; link next; };
link head, z, t;
void listinitialize() {
   head=malloc(sizeof *head); z=malloc(sizeof *z);
   head->next = z; z->next = z;
}
/* node deallocation part is not implemented here */
void deletenext(link t) { t->next = t->next->next; }
link insertafter(int v, link t) {
   link x;
   x = malloc(sizeof *x);
   x->key = v; x->next = t->next; t->next = x;
   return x;
}
```

配列によるリストの表現

```
int key[MAX+2], next [MAX+2];
int head, z, x;
void listinitialize() {
   head = 0; z = 1; x =2; //xはノードの数を示す
   next[head] = z; next[z] = z;
}
void deletenext(int t) {
   next[t] = next[next[t]];
}
link insertafter(int v, int t) {
   key[x] = v; next[x] = next[t]; next[t] = x;
   return x++;
}
```

注）：ここで、削除された要素の再利用について考えていない。

List implemented by array

```
int key[MAX+2], next [MAX+2];
int head, z, x;
void listinitialize() {
   head = 0; z = 1; x =2; // x: node number
   next[head] = z; next[z] = z;
}
void deletenext(int t) {
   next[t] = next[next[t]];
}
link insertafter(int v, int t) {
   key[x] = v; next[x] = next[t]; next[t] = x;
   return x++;
}
```

Note: here, no care about the reuse of deleted elements.

リスト・配列の使い分け

- データの最大の大きさが前もって予想でき、頻繁に挿入や削除しない場合は、配列による表現がよい
  - 配列の簡単な構造により効率を上げることができる
- そうでない場合は、連結リストで実現した方がよい
  - リストはメモリ上同一ブロック内になくてもよいので、メモリを自由に増やすことができる
  - 削除や要素の順番の変更がリンクの変更だけでできるので、効率がよい

List or array?

- Array is usually faster than list, if we can predict the maximum size of data, and no need to insert or delete elements frequently inside of the data block.
- List is more flexible than array. Each element of list can be allocated separately everywhere in the memory space. Insertion and deletion of elements is fast and easy by changing their links.

双方向リスト(doubly kinked list)

- 与えられた要素の前の要素にアクセスためにはどうすればいいか？
  - 一般的にはリストは先頭からのアクセスしかサポートしない
  - 前の要素にアクセスするためには前の要素へのポインタをセーブするか、また先頭から順にアクセスするしかない。
- 双方向リスト
  - 各ノードに順方向へのリンクと逆方向へのリンクを持つリスト。

Doubly kinked list

- Doubly linked list is double sided linked list.
- Differing to singly linked list, a doubly linked list has a link to the previous node as well as to the next node.

来週のトピックス

- 基本データ構造（その２）
  - スタック
  - キュー
  - 抽象データ型

Topics of next week

- Basic data structures (part 2)
  - stack
  - queue
  - abstract data type