Today's contents

- Structure
  - structure, union
- Linear list
- Binary tree
- Hash
  - Hash function
  - Chained hash

今日の内容

- 構造体
  - 構造体、共用体
- 線形リスト
- ２分木
- ハッシュ
  - ハッシュ関数
  - チェイン法

Structure

- Array
  - Gathering and treating plural data
  - But, only for the same type data

- Structure
  - Gathering and treating different type data
  - You can also make an array of structure.

構造体

- 配列
  - 複数のデータをまとめて扱うことができる
  - ただし、同じ型のデータだけ
- 構造体
  - 異なるデータ型をまとめて扱う
  - 構造体自体を配列にすることもできる

Declaration of structure template

- Declare a structure template combining plural data types (structure member)
  - This is just “declaration” (not “definition”) so that the real memory area is still not allocated.

≪Format≫

```
struct Identifier_Name {
  Data_type1 Variable_name1;
  Data_type2 Variable_name1;
  ...
  Data_typeN Variable_nameN;
};
```

≪Example≫

```
struct Student {
  int gender;            // 0:Male, 1:Female
  char id[10];           // Student ID
  char name[20];
  char bloodType[3];
  double weight;
  double height;
};
```

構造体テンプレートの宣言

- 複数のデータ(構造体メンバー)をまとめて、1つの構造体テンプレート(構造体のデータ構造)を宣言する
  - 「宣言」であるため、まだ実体はない(「定義」ではない)

≪書き方≫

```
struct 構造体識別子 {
  データ型1 変数名1;
  データ型2 変数名2;
  ...
  データ型n 変数名n;
};
```

≪例≫

```
struct Student {
  int gender;              // 性別 0:男, 1:女
  char id[10];             // 学籍番号
  char name[20];           // 氏名
  char bloodType[3];       // 血液型
  double weight;           // 体重
  double height;           // 身長
};
```

Definition of structure variable

- By using structure template, define an actual structure variable (allocate actual memory for the variable)
  - The scope of a structure template is the same as for variables.

≪Format≫

```
struct  Identifier_Name  Variable_name1, Variable_name2, ....., Variable_nameN;
```

≪例≫

```
struct  Student  student1;          // define the structure variable named “student1”
struct  Student  student2[20];      // define the array of structure named “student2”
```

構造体変数の定義

- 構造体テンプレートを使って、構造体変数の実体を定義（実際にメモリを確保）
  - 構造体テンプレートの有効範囲は変数のスコープと同じ

≪書き方≫

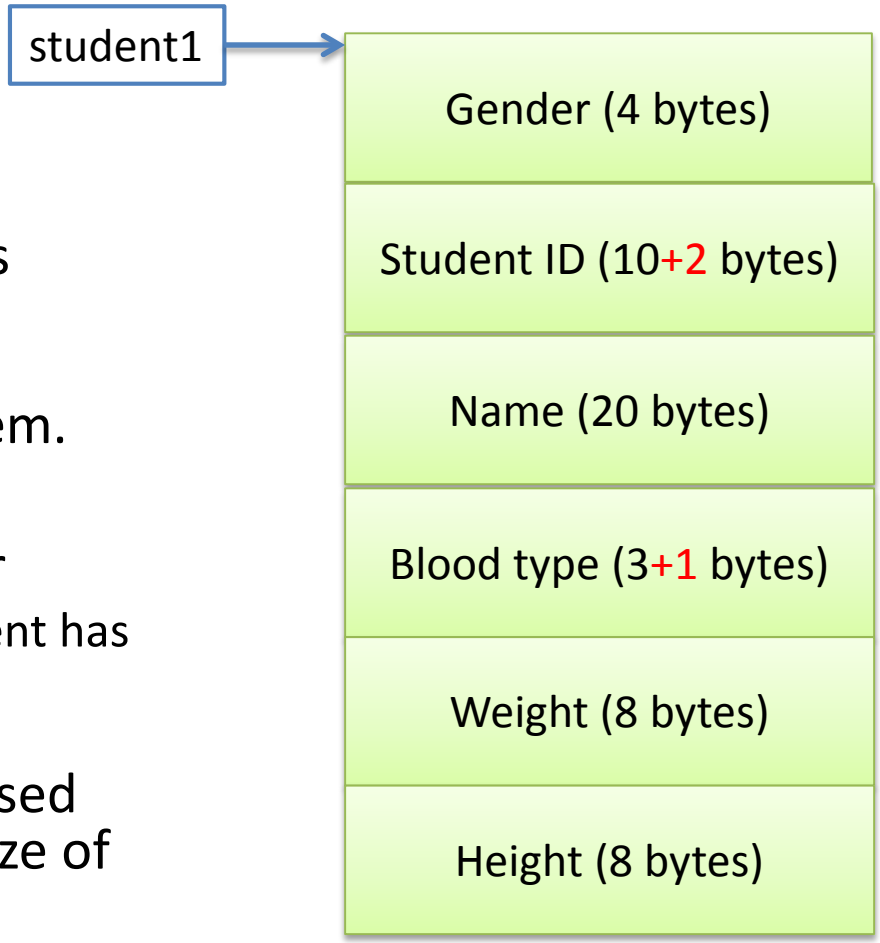
```
struct　構造体識別子　構造体変数名１，構造体変数名２，……，構造体変数名ｎ；
```

≪例≫

```
struct   Student   student1;          // student1という名前の構造体変数を定義
struct   Student   student2[20];      // student2という名前の構造体配列を定義
```

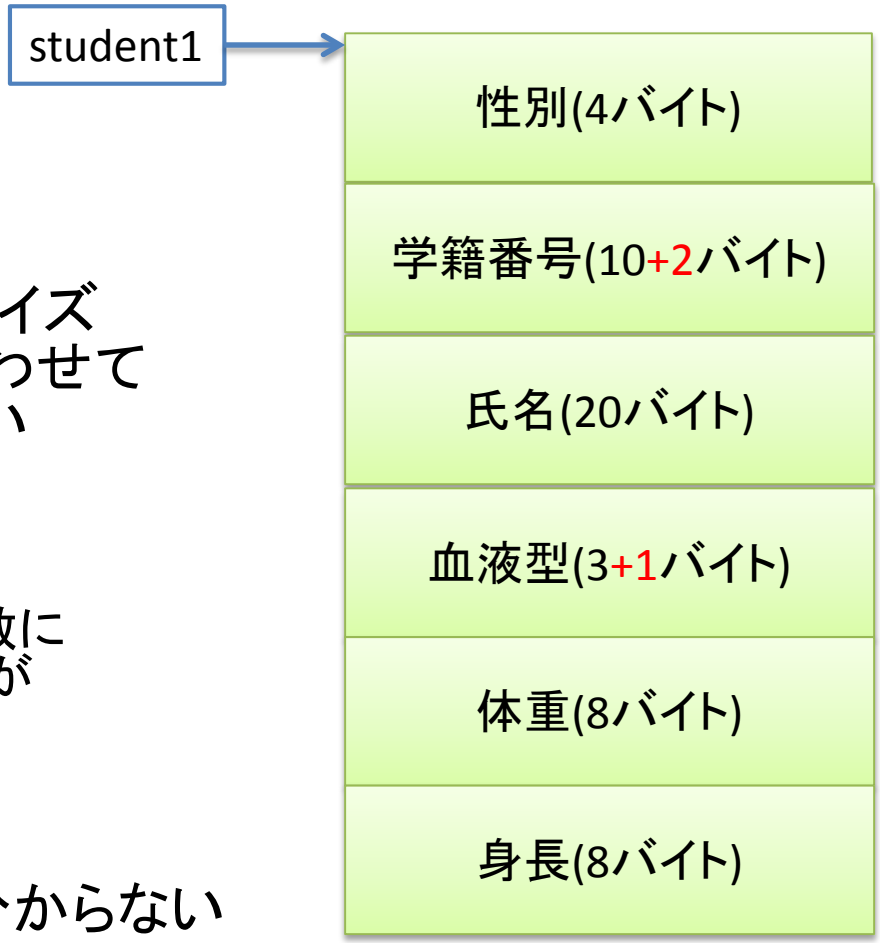
Structure and memory alignment (1)

- Actual memory allocation depends on a computer system.
- Usually, the memory allocation is done according to “word”, the byte size which is easy to be processed by the computer system.
- Figure: Example of 4-byte border
  - Use “pads” so that each component has multiple bytes of 4.
- “sizeof(struct Student)” will be used to know the actual whole byte size of the structure including pads.



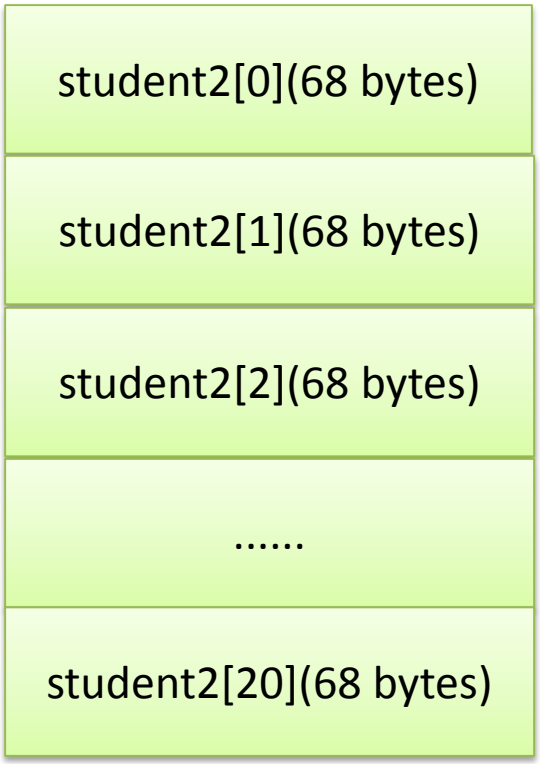
構造体とメモリ配置（1）

- メモリ上にどのように構造体が配置されるかは、処理系に依存する
- 計算機の処理しやすいバイトサイズ（ワード）があるため、それに合わせてメモリ配置が行われることが多い
- 右の図は4バイト境界の例
  - それぞれの要素が4バイトの倍数になるように、パディング（詰め物）が入れられている
- 実際の構造体全体のサイズは、sizeof(struct Student) でないと分からない



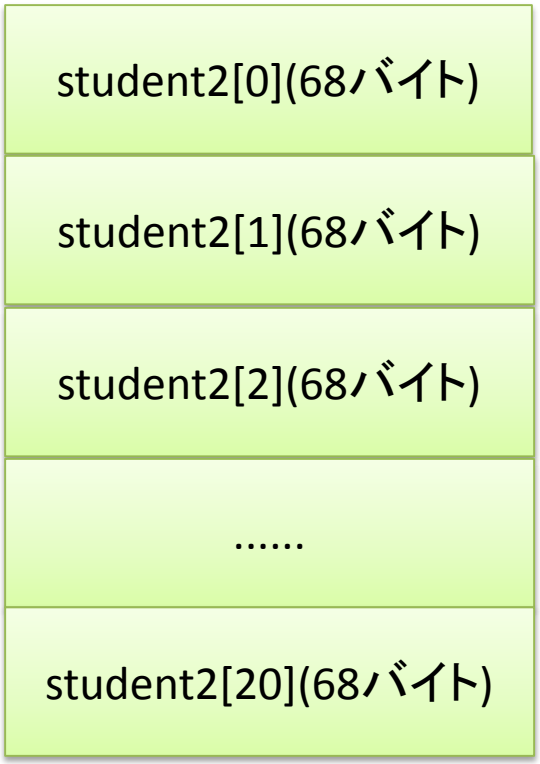
Structure and memory alignment (2)

- Former example
  - “sizeof(struct Student)” is 68 bytes.
- An array of structure has a form which aligns each structure in line on the memory.



構造体とメモリ配置（2）

- 先ほどの例
  - sizeof(struct Student)は68バイト
- 構造体配列は、パディングされた構造体一つ一つがメモリ上に順番に配置される形となる



Initialization of structure

- Initialization of structure variables (members)

```
struct   Student   student1 = { 0, "B0001",“Taro Yamada", "A", 72.5, 170.0 };
```

- Initialization of an array of structure variables

```
struct   Student   student2[20]   =   {
    { 0, "B0003", “Jiro Tanaka", "AB", 55.9, 168.6 },
    { 1, "B0009", “Hanako Suzuki", "O", 45.0, 157.2 },
    { 0, “B0015”, “Saburo Sato”, “B”, 85.2, 180.2 },
};
```

  - Write as the same as the initialization of an array
  - Write members in order as the declaration

構造体の初期化

- 構造体変数の初期化

```
struct  Student  student1 = { 0, "B0001","山田太郎", "A", 72.5, 170.0 };
```

- 構造体配列の初期化

```
struct  Student  student2[20]  =  {
    { 0, "B0003", "田中二郎", "AB", 55.9, 168.6 },
    { 1, "B0009", "鈴木花子", "O", 45.0, 157.2 },
    { 0, “B0015”, “佐藤三郎”, “B”, 85.2, 180.2 },
};
```

  - 配列の初期化と同じような感じで
  - 宣言通りのメンバーの順序で書く

Referring a structure

- Use “.”（dot operator）
  - 「(structure_name).(member_name)」
- Referring a structure variable

```
printf("%d %s %s %s %3.1f %4.1f¥n¥n",
    student1.gender, student1.id, student1.name, student1.bloodType,
    student1.weight, student1.height);
```

- Referring an array of structures

```
for (i = 0; i < 3 ; i++ ) {
    printf("%d %s %s %s %3.1f %4.1f¥n",
        student2[i].gender, student2[i].id,
        student2[i].name, student2[i].bloodType,
        student2[i].weight, student2[i].height);
}
```

構造体の参照

- .（ドット演算子）を用いる
  - 「（構造体名）.（メンバー名）」
- 構造体変数の参照

```
printf("%d %s %s %s %3.1f %4.1f¥n¥n",
    student1.gender, student1.id, student1.name, student1.bloodType,
    student1.weight, student1.height);
```

- 構造体配列の参照

```
for (i = 0; i < 3 ; i++ ) {
    printf("%d %s %s %s %3.1f %4.1f¥n",
        student2[i].gender, student2[i].id,
        student2[i].name, student2[i].bloodType,
        student2[i].weight, student2[i].height);
}
```

Structure and function (1)

- Passing a structure to a function
  - function(student1);
    - “Call by Value” ⇒ Passing a copy of the structure to the function
    - How to refer a structure: “.” (dot operator) (the same as the former example)
  - function(&student1);
    - “Call by Value” of an address ⇒ Passing an address of the structure to the function
    - How to refer a structure: “->” (arrow operator)
      - 「(pointer_to_structure)->(member_name)」
      - Refer a member without a pointer operator “*”.

構造体と関数（1）

- 構造体を関数に渡す
  - function(student1);
    - 値渡し ⇒ 構造体のコピーが関数に渡される
    - 関数内での構造体の参照は先ほどの例と同じ（ドット演算子）
  - function(&student1);
    - アドレス渡し ⇒ 構造体のアドレスが関数に渡される
    - 関数内での構造体の参照には、”->”（アロー演算子）を用いる
      - 「（構造体へのポインタ）->（メンバー名）」
      - ポインタ演算子（*）を使わないで書ける

Structure and function (2)

- How to use an arrow operator

```
int   main()
{
    struct   Student   student1;

    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        student1.gender, student1.id, student1.name, student1.bloodType,
        student1.weight, student1.height);

    function( &student1 );
}

void function( struct  Student  *p )
{
    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        p->gender, p->id, p->name, p->bloodType, p->weight, p->height);
}
```

構造体と関数（2）

- アロー演算子の使い方

```
int   main()
{
    struct   Student   student1;

    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        student1.gender, student1.id, student1.name, student1.bloodType,
        student1.weight, student1.height);

    function( &student1 );
}

void function( struct  Student  *p )
{
    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        p->gender, p->id, p->name, p->bloodType, p->weight, p->height);
}
```

Structure and union

- union
  - Similar declaration as structure
  - Members share the memory
    - The union size is a maximum size among members.
  - Union seems like "structure with zero offset".
    - For "struct", members are allocated on the memory in line.

≪Format≫

```
union   Identifier_Name   {
   DataType1   VariableName1;
   DataType2   VariableName2;
   ...
   DataTypeN VariableNameN;
  } ;
```

構造体と共用体

- 共用体 union
  - 構造体と同じような宣言
  - メンバーがメモリを共有する
    - サイズは、最大のバイト数を持つメンバーのサイズとなる
  - 共用体は「オフセットがゼロの構造体」と見ることもできる
    - 構造体では、メンバーがメモリ上に並ぶ

≪書き方≫

```
union　共用体識別子　{
  データ型１　変数名１;
  データ型２　変数名２;
  …
  データ型n　変数名n;
};
```

typedef and structure

- Writing “struct Student” bothers me.
- Writing that the easy way by “typedef”

≪Example≫

```
typedef struct Student {
    int gender;             char id[10];
    char name[20];          char bloodType[3];
    double weight;          double height;
} STUDENT, *STUDENTP;

int main()
{
    STUDENT student1;  // the same as “struct Student student1;”
    STUDENT *p1;       // the same as “struct Student *p1;”
    STUDENTP p2;       // the same as “struct Student *p2;”

    p2 = p1 = &student1;
}
```

typedefと構造体

- いちいち、”struct Student” と書くのが面倒だ
- typedef を使って楽をする

≪例≫

```
typedef struct Student {
    int gender;          char id[10];
    char name[20];       char bloodType[3];
    double weight;       double height;
} STUDENT, *STUDENTP;

int main()
{
    STUDENT student1; // “struct Student student1;”と同義
    STUDENT *p1;      // “struct Student *p1;”と同義
    STUDENTP p2;      // “struct Student *p2;”と同義

    p2 = p1 = &student1;
}
```

self-reference structure

- Structure which has a pointer to the own structure in the structure declaration
  - It's just a pointer to a structure variable, not a pointer to an own instance of the structure.

```
struct KeyValueList {
    int key;
    double value;
    struct KeyValueList *next; // self-reference
};
```

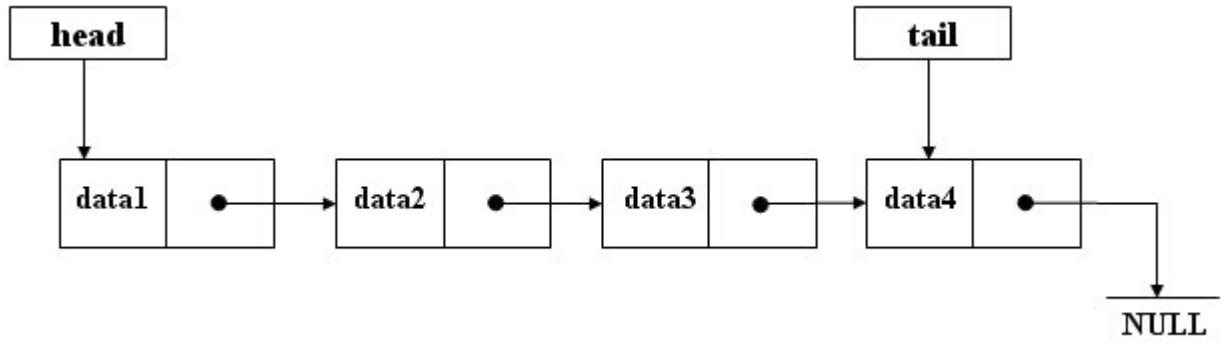
自己参照型構造体

- 構造体宣言の中で自らの構造体へのポインタを持つような構造体
  - 「構造」へのポインタであって、インスタンスとしての自分自身ではないので、混同しないこと

```
struct KeyValueList {
    int key;
    double value;
    struct KeyValueList *next; // ここが自己参照
};
```

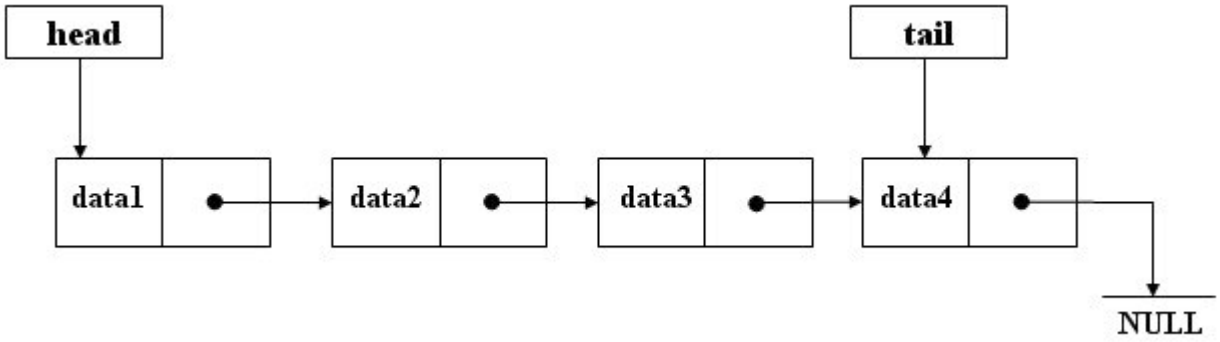
Linear list

- The simplest linked list
  - unidirectional list (singly linked list), this time
  - some times, bidirectional list (doubly linked list) which has reverse links
- Each node has a link to the next node. The final node of the list has NULL as the link.



線形リスト

- 連結リストの中でも、最も単純なもの
  - 今回は片方向リスト
  - 後ろ向きのリンクも持たせて双方向とする場合もある
- ノード毎に1つのリンクを持ち、このリンクがリスト上の次のノードを指す。リストの最後尾ならNULL値を格納する。



Insertion into linear list (1)

- Memory allocation for a list component
  - Use malloc() to allocate a struct memory area.

```
typedef  struct  KeyValueList  {
    int  key;
    double  value;
    struct  KeyValueList  *next;
}  LIST,  *LISTP;

int  main()
{
    LISTP  p  =  ( LISTP )malloc( sizeof( LIST ) );
    p->key  =  1;
    p->value  =  10.0;
    p->next  =  NULL;
}
```

線形リストへの挿入（1）

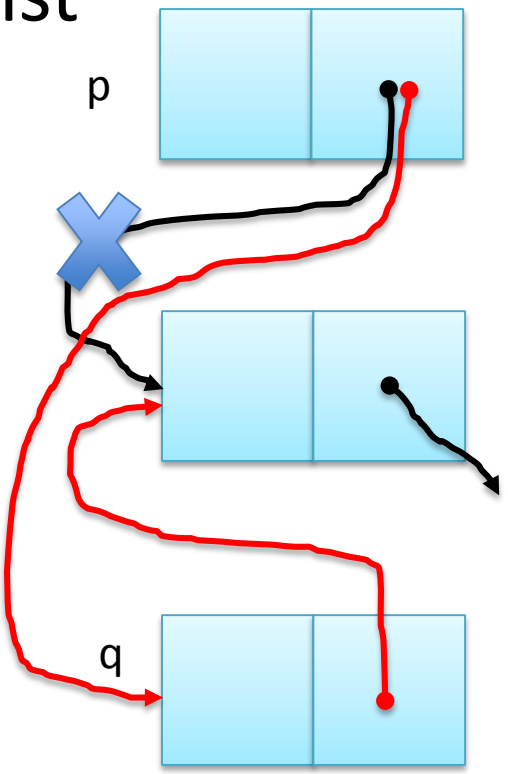
- リストの要素領域の確保
  - malloc()で構造体領域を確保

```
typedef  struct  KeyValueList  {
    int  key;
    double  value;
    struct  KeyValueList  *next;
}  LIST,  *LISTP;

int  main()
{
    LISTP  p  =  ( LISTP )malloc( sizeof( LIST ) );
    p->key  =  1;
    p->value  =  10.0;
    p->next  =  NULL;
}
```

Insertion into linear list (2)

- Insertion of a component into the list
  - Insert q into the place indicated by p

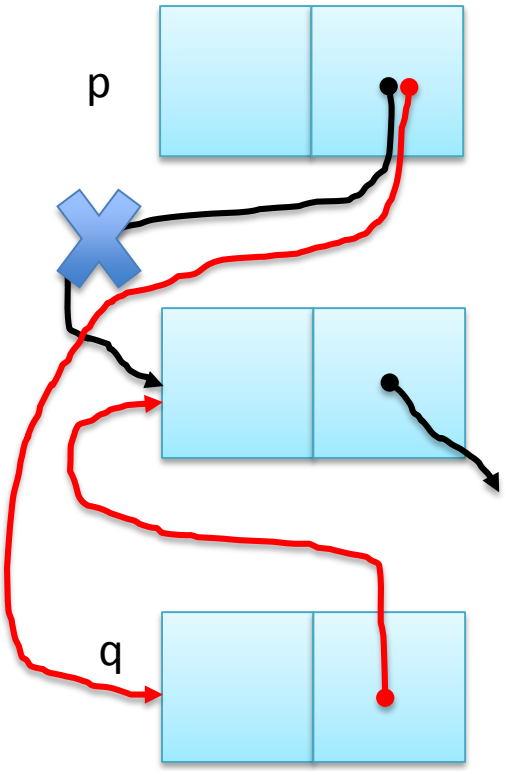
```
// Memory allocation
LISTP   q   =   ( LISTP )malloc( sizeof( LIST ) );

// Assigning values
q->key  =  1;
q->value  =  10.0;

// Change the link
q->next  =  p->next;
p->next  =  q;
```



線形リストへの挿入(2)

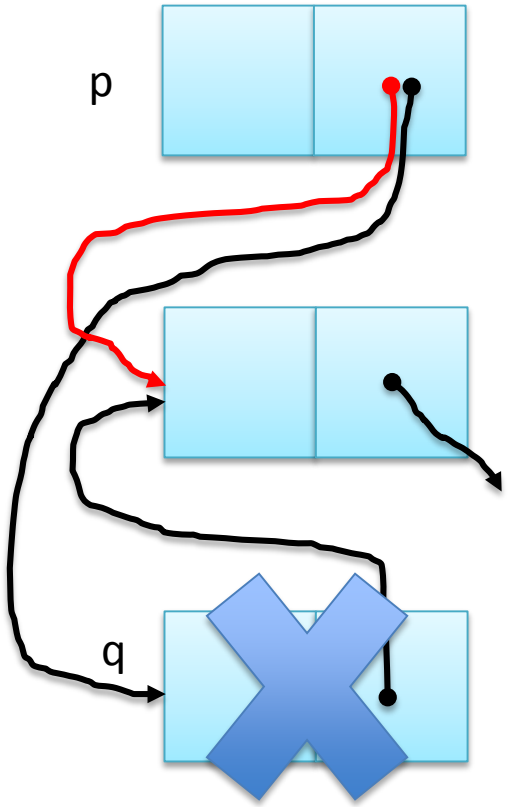
- リストへの要素の挿入
  - p が指す位置へ q を挿入する

```
// 領域確保
LISTP   q   =   ( LISTP )malloc( sizeof( LIST ) );

// 値を入れる
q->key  =  1;
q->value  =  10.0;

// リンク先の変更
q->next  =  p->next;
p->next  =  q;
```



Deletion from linear list (1)

- Delete a component from the list
  - Delete the component q next to p



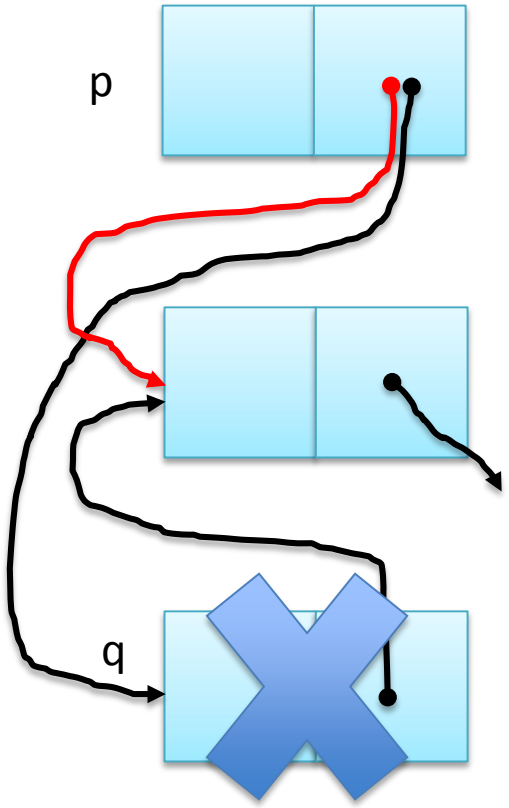

```
LISTP   q  =  p->next;

// Change the link
p->next  =  q->next;

// Free the memory
free(q);
```

  - "Segmentation Fault" if you free the memory in first.

線形リストからの削除（1）

- リストから要素を削除
  - p の次の要素 q を削除する

```
LISTP  q  =  p->next;

// リンク先の変更
p->next  =  q->next;

// メモリ解放
free(q);
```



  - 先にメモリを解放するとエラーになるので注意

Deletion from linear list (2)

- Delete the whole list

```
void   deleteMemory( LISTP p )
{
    if   ( p != NULL ) {
         deleteMemory( p->next );
         free( p );
    }
    return;
}
```

  - Delete the list by recursion
  - Call the function from “head”, but free() will be done from the tail.

線形リストからの削除（2）

- リスト全体の削除

```
void   deleteMemory( LISTP p )
{
    if   ( p != NULL ) {
         deleteMemory( p->next );
         free( p );
    }
    return;
}
```

  - 再帰呼び出しにより削除を行う
  - 通常 head から呼ぶが、実際に free() が行われるのは末尾からとなる

Displaying linear list

- Follow the list from "head"
- The end of the list when NULL comes.
- You can write this by using recursion.

```
void  printNode( LISTP p )
{
   printf( " %d: %f ", p->key, p->value );
}

void printList( LISTP head )
{
   LISTP  p  =  head;
   int  i  =  1;
   while  ( p  !=  NULL )  {
      printf( "%d-th data is ",  i );
      printNode( p );
      printf( ".¥n");
      p  =  p->next;  i++;
   }
}
```

線形リストの表示

- head から順番にリストを辿って行く
- NULLがきたらリストの末尾
- 再帰で書くこともできる

```
void  printNode( LISTP p )
{
   printf( " %d: %f ", p->key, p->value );
}

void printList( LISTP head )
{
   LISTP  p  =  head;
   int  i  =  1;
   while  ( p  !=  NULL )  {
      printf( "%d番目のデータは ",  i );
      printNode( p );
      printf( "です。¥n");
      p  =  p->next;  i++;
   }
}
```

Binary tree (1)

- A tree structure with root which each node has 2 children at most.

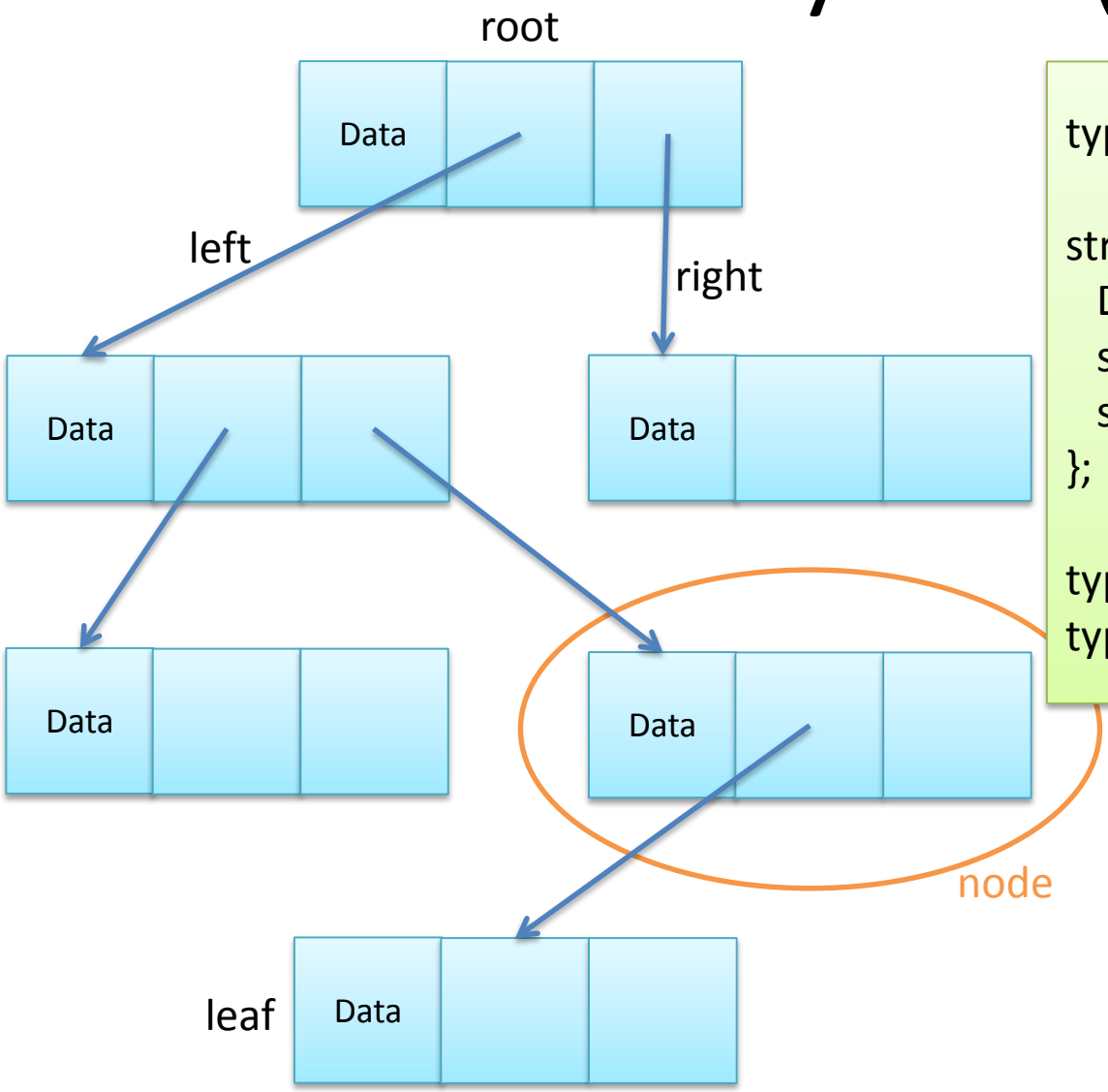
- The data structure of binary tree starts root of “tree”, and has “left subtree” and “right subtree” logically.
- Usually, a “node of tree” has data. This data is a kind of structure so that the data are heterogeneous. One of the data is “key” which represents magnitude relationships.

２分木（１）

- 根（root）付き木構造の中で、あるノード（node）が持つ子の数が高々２であるもの

- ２分木のデータ構造は、「木」の根から始まり、「左部分木」と「右部分木」の２つが理論的に存在する
- 通常、「木のノード」にはデータが保持される。このデータは一種の構造体であり、異種のデータが保持される。その中に「キー」と呼ばれるデータがあり、キーによって大小判定が行われる

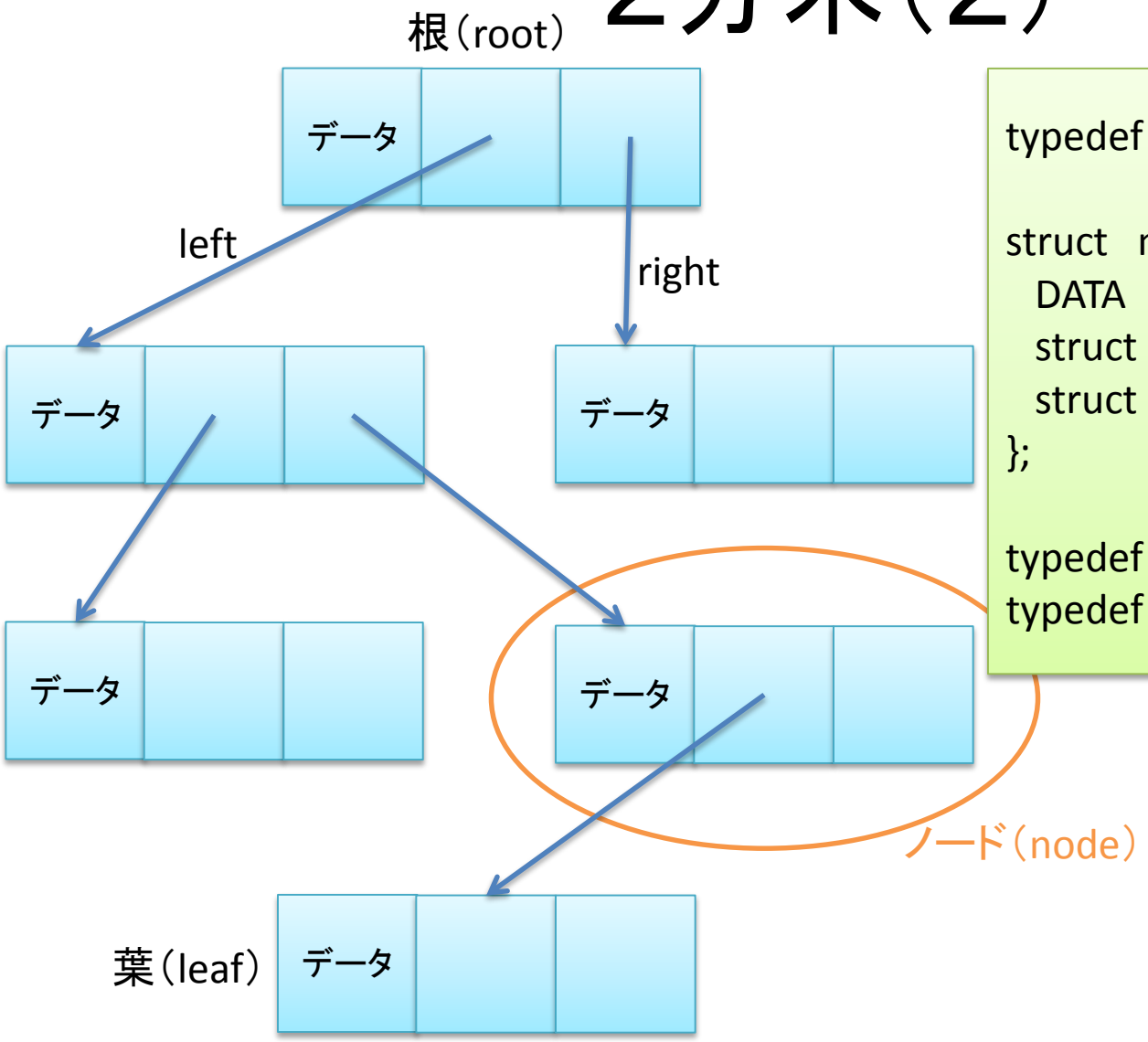
Binary tree (2)




```
typedef  char  DATA;

struct  node  {
  DATA  d;
  struct  node  *left;    // subtree
  struct  node  *right;  // subtree
};

typedef  struct  node  NODE;
typedef  NODE  *BTREE;
```

The node has a link to the parent, in some case.

2分木(2)



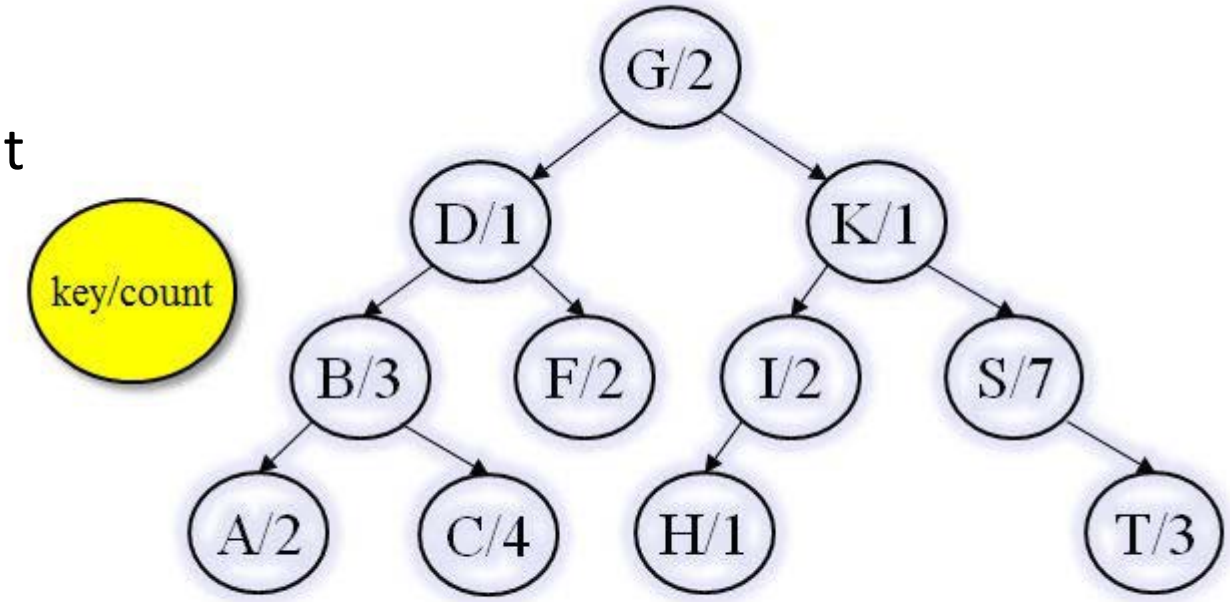

```
typedef  char  DATA;

struct  node  {
  DATA  d;
  struct  node  *left;    // 左部分木
  struct  node  *right;   // 右部分木
};

typedef  struct  node  NODE;
typedef  NODE  *BTREE;
```

親へのリンクを
持たせる場合もある

Binary tree (3)

- Given a character string “GDKSIFSFBGSASCAHITSTSBCBCSCT”, insert each character to a binary tree from the left.
  - Insert left if the character is smaller than the parent node. Insert right if the character is bigger than the parent node.
  - If the node exists already, count it up.

- Result

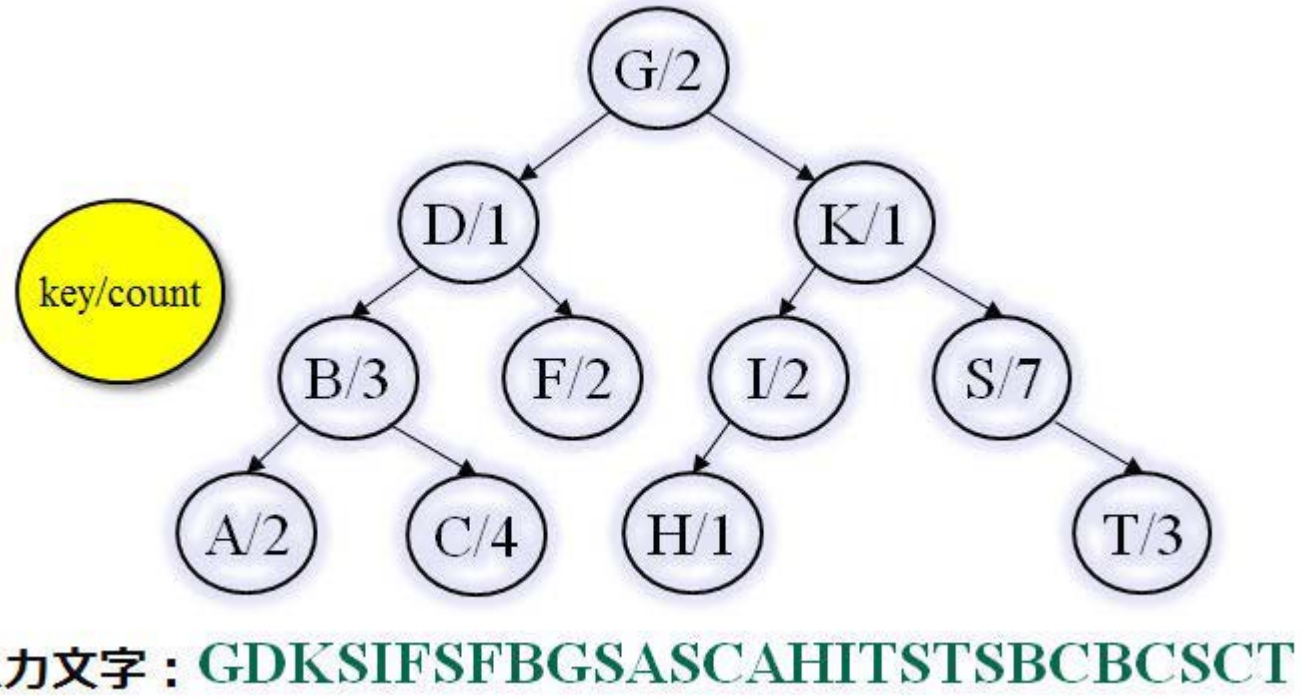


Input: GDKSIFSFBGSASCAHITSTSBCBCSCT

2分木（3）

- 「GDKSIFSFBGSASCAHITSTSBCBCSCT」という文字列が与えられたとき、文字列から1文字ずつ2分木のノードに挿入する
  - 親ノードより前の文字なら左に、後の文字なら右に入れることにする
  - 既に文字のノードが存在する場合はカウントアップする

- 結果



Insertion into binary tree

```
BTREE  insertNode( BTREE  node,  DATA  x )
{
   BTREE  p = node;

   if ( p == NULL ) {  // Insert to root
      p = allocNode( x );
      return  p ;
   }
   if ( x  ==  p->d ) {
      // Process in the equal case.
   } else if ( x < p->d ) {  // Insert into left subtree
      p->left = insertNode(p->left, x);    // Recursion
   }  else  {          // Insert into right subtree
      p->right = insertNode(p->right, x);    // Recursion
   }
   return  p ;
}
```

Insert the data so that the left child is bigger than the parent and right child is smaller than the parent. (for binary search)

```
BTREE  allocNode( DATA  x )
{
   BTREE  p;

   p =  (BTREE)malloc(sizeof(NODE));
   p->d  =  x;
   p->left  =  p->right  =  NULL;

   return  p ;
}
```

2分木への挿入

```
BTREE  insertNode( BTREE  node,  DATA  x )
{
   BTREE  p = node;

   if ( p == NULL ) {  // rootに挿入
      p = allocNode( x );
      return  p ;
   }
   if ( x  ==  p->d ) {
      // 等しいときの処理
   } else if ( x < p->d ) {  // 左部分木に割り当て
      p->left = insertNode(p->left, x);    // 再帰呼出し
   }  else  {          // 右部分木に割り当て
      p->right = insertNode(p->right, x);    // 再帰呼出し
   }
   return  p ;
}
```

左の子は親より小さく、
右の子は親より大きくなるように
データの挿入を行うことにする
（2分探索木）

```
BTREE  allocNode( DATA  x )
{
   BTREE  p;

   p =  (BTREE)malloc(sizeof(NODE));
   p->d  =  x;
   p->left  =  p->right  =  NULL;

   return  p ;
}
```

Recursion of binary tree

- Order
  - Pre-order
    - Current node ⇒ Left subtree ⇒ Right subtree
    - “On-the-way order”
  - In-order
    - Left subtree ⇒ Current node ⇒ Right subtree
    - “In-passing order”
  - Post-order
    - Left subtree ⇒ Right subtree ⇒ Current node
    - “On-the-way-back order”

2分木の再帰的呼び出し

- 順序
  - 前順序（pre-order）
    - 今のノードの処理 ⇒ 左部分木 ⇒ 右部分木
    - 「行きがけ順」
  - 中順序（in-order）
    - 左部分木 ⇒ 今のノードの処理⇒ 右部分木
    - 「通りがけ順」
  - 後順序（post-order）
    - 左部分木 ⇒ 右部分木 ⇒今のノードの処理
    - 「帰りがけ順」

Sorting binary tree elements

- By using “In-order”, the result is sorted automatically, because the data were inserted so that the left child is bigger than the parent and right child is smaller than the parent.

```
void printBtree( BTREE p )
{
   if (p != NULL ) {
      printBtree( p->left );       //  Left subtree (recursion)
      printf( “%s¥n”,  p->d );
      printBtree( p->right );     //  Right subtree (recursion)
   }
   return;
}
```

２分木のソート表示

- 左の子は親より小さく、右の子は親より大きくなるようにしているので、中順序で表示すれば、自動的にソートされた結果となる

```
void printBtree( BTREE p )
{
   if (p != NULL ) {
      printBtree( p->left );       // 左部分木の処理（再帰呼び出し）
      printf( “%s¥n”,  p->d );
      printBtree( p->right );     //  右部分木の処理（再帰呼び出し）
   }
   return;
}
```

Free binary tree

- Use “Post-order”, not to free pointers in first which should be accessed.
  - If p is freed in first, p->left and p->right cannot be accessed after that.

```
void freeBtree( BTREE p )
{
   if (p != NULL ) {
      freeBtree( p->left );       //  Left subtree (recursion)
      freeBtree( p->right );     //  Right subtree (recursion)
      free( p );
   }
   return;
}
```

２分木の解放

- アクセスすべきポインタを先に解放してしまわないように、後順序で解放していく
  - pを解放してしまうと、その時点でp->leftやp->rightにアクセスできなくなる

```
void freeBtree( BTREE p )
{
   if (p != NULL ) {
      freeBtree( p->left );      // 左部分木の処理（再帰呼び出し）
      freeBtree( p->right );    //  右部分木の処理（再帰呼び出し）
      free( p );
   }
   return;
}
```

Hash (1)

- Hash search
  - A search algorithm with *O(1)*
  - Data are inserted in an array (hash table).
    - Each component of a hash table is called “bucket”.
  - Insertion place is decided uniquely by some sort of operation (hash function) for the key of data.
    - The value calculated by hash function is called “hash value”.
  - Hash search is done by getting data at the place of a hash value calculated from the key in the hash table.
    - It’s *O(1)* because of simple table look-up.

ハッシュ（１）

- ハッシュ探索
  - 探索を*O(1)*で行うアルゴリズム
  - データは配列（ハッシュ表）に登録される
    - ハッシュ表の各要素を「バケット」と呼ぶ
  - 登録位置は、データに関するキーに対して、何らかの演算（ハッシュ関数）を施すことで一意に決定される
    - ハッシュ関数で求まる値を「ハッシュ値」という
  - 探索は、キーからハッシュ値を求め、ハッシュ表の中のハッシュ値の位置にあるデータを得ることで行われる
    - 単なるテーブルルックアップのため *O(1)* となる

Hash (2)

- Condition of *O(1)* hash
  - Can calculate an unique hash value for a key
    - If hash values are overlapped, different data should be stored into the same bucket. It's not *O(1).*
  - Hash table is bigger than the data should be stored.
    - If the hash table is smaller than the number of data, some of hash values would be overlapped. (Hash Collision)
    - Actually, it's difficult to avoid the hash collision, so it's hoped that the hash table is bigger enough than the number of data.

ハッシュ（2）

- ハッシュが*O(1)*になる条件
  - キーに対して、ユニークなハッシュ値が計算できること
    - ハッシュ値が重複すると同じバケットに異なるデータを格納しなければならなくなるため、*O(1)*ではなくなる
  - 格納するデータ数よりもハッシュ表が大きいこと
    - データ数よりもハッシュ表が小さければ、ハッシュ値が重複する（衝突）
    - 実際には、ハッシュ値の重複は避けられないので、ハッシュ表はデータ数よりも十分に大きいことが望ましい

Hash (3)

- Hash function
  - A function to calculate a hash value
  - The hash value will be an index for the hash table (array)
  - Insert data “v” for key “a”
    - Hash function : hash()
    - Hash table : htable[]
    - Data for key “a” : htable[hash(“a”)] = v;

    - If the key is a positive number, you can use it as a hash value.
    - If the key is a character string, etc., you should exchange it to a unique value somehow.

ハッシュ（3）

- ハッシュ関数
  - ハッシュ値を算出する関数
  - ハッシュ値がハッシュ表（配列）に対するインデックスとなる
  - キー ”a” に対するデータ “v” をハッシュ表に登録する
    - ハッシュ関数 : hash()
    - ハッシュ表 : htable[]
    - キー “a” に対するデータ : htable[hash(“a”)] = v;

    - キーが正数値なら、それをそのままハッシュ値としても良い
    - キーが文字列等であれば、どうにかしてユニークな数値に変換する必要がある

Hash function (1)

- Good hash function
  - can calculate a hash value which does not overlap other hash values.
  - Actually, it's very difficult.

- Example of simple hash function
  - calculate a hash value by adding all of the character code in the key (a character string) and divide it by 100.
    - The size of hash table is 100.

```
int   hash( char *s )
}
    int   i = 0;
    while   ( *s )   i += *s;
    return   i % 100;
}
```

ハッシュ関数（1）

- 良いハッシュ関数
  - ハッシュ値がハッシュ表の大きさに対して重複しないように計算できる
  - 実際はかなり難しい

- 簡単なハッシュ関数の例
  - キー（文字列）に入っている文字の文字コードを全て加算し、100で割った余りをハッシュ値とする
    - ハッシュ表の大きさが100

```
int   hash( char *s )
}
    int   i = 0;
    while   ( *s )   i += *s;
    return   i % 100;
}
```

Hash function (2)

- Former example
  - works reasonably well
  - Collision happens
    - "one" and "neo" have the same hash value. ⇒ Collision
      - The same characters are used for composing the character strings.
    - "five" and "nine" have the same hash value.
      - Collision happens for complete different character strings.
  - Improvement idea
    - Use variable weights depending on what number the character is.
    - Let the hash table size be a prime number.

```
int   hash( char *s )
}
    int   i = 0;
    while   ( *s )   i += *s;
    return   i % 100;
}
```

```
int   hash( char *s )
}
    int   i = 0,   j = 1;
    while   ( *s )   {
        i = i * 26 + (*s – 'A');
        // Warning: Overflow!!
    }
    return   i % 101;
}
```

ハッシュ関数（2）

- 先ほどの例
  - そこそこ上手く動く
  - 衝突は起きる
    - “one”と“neo”は同じハッシュ値となるので、衝突する
      - 文字列を構成する文字が同じ
    - “five” と“nine”も同じハッシュ値
      - 文字列が全然違っていても衝突は起きる
  - 改良案
    - 何文字目かで重みを変える
    - ハッシュ表の大きさを素数にする

```
int   hash( char *s )
}
    int   i = 0;
    while   ( *s )   i += *s;
    return   i % 100;
}
```

```
int   hash( char *s )
}
    int   i = 0,   j = 1;
    while   ( *s )   {
        i = i * 26 + (*s – ‘A’);
        // オーバーフロー注意
    }
    return   i % 101;
}
```

If collided…

- Want to avoid “hash collision”
  - Make the hash table big
  - Use a good hash function

  - Nevertheless, a collision happens !

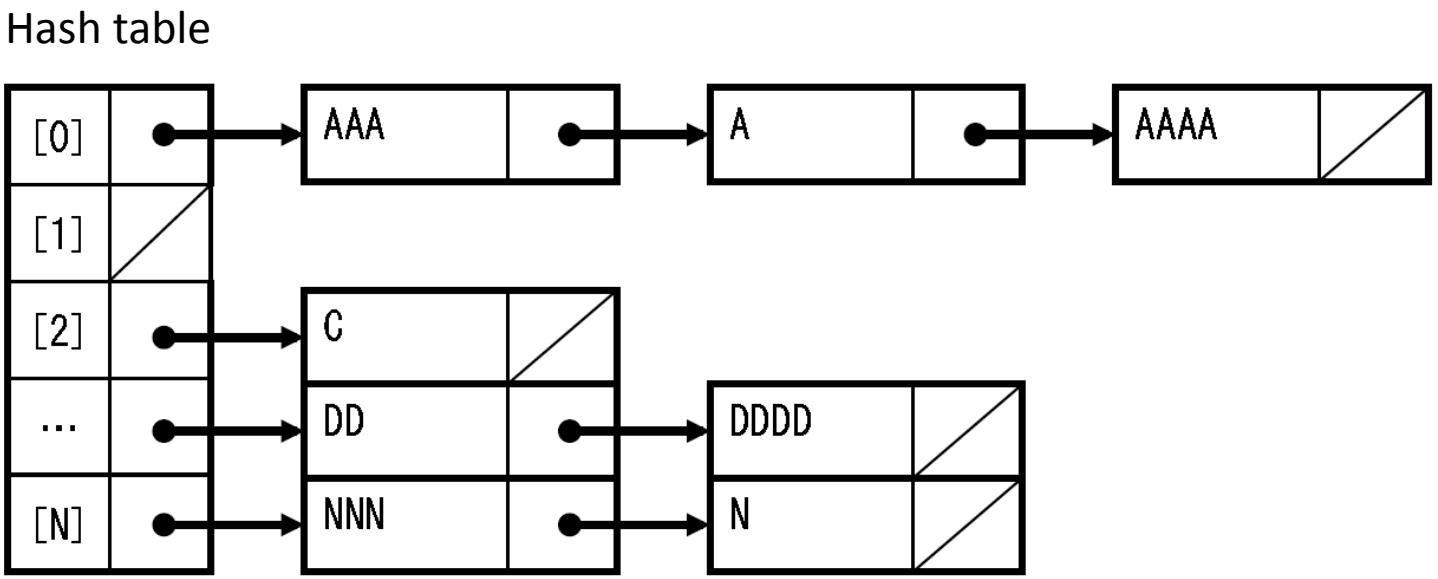
- How to go ?
  - Open address hash
    - Use another hash function to get another hash value (search another place in the hash table to store the data) ⇒ re-hash
    - k-th re-hash hash(x, k) = (hash(x) + k) % HASHSIZE; etc.
  - Chained hash
    - Next page

衝突したら……

- 衝突を避けたい
  - ハッシュ表を大きくする
  - 良いハッシュ関数を用いる
  - それでも衝突しちゃう！
- どうするか？
  - オープンアドレス法
    - 別のハッシュ関数を使って、違うハッシュ値を求めて使う（ハッシュ表の違う格納位置を探す）⇒ 再ハッシュ
    - k 回目の再ハッシュ　hash(x, k) = (hash(x) + k) % HASHSIZE; 等
  - チェイン法
    - 次ページ

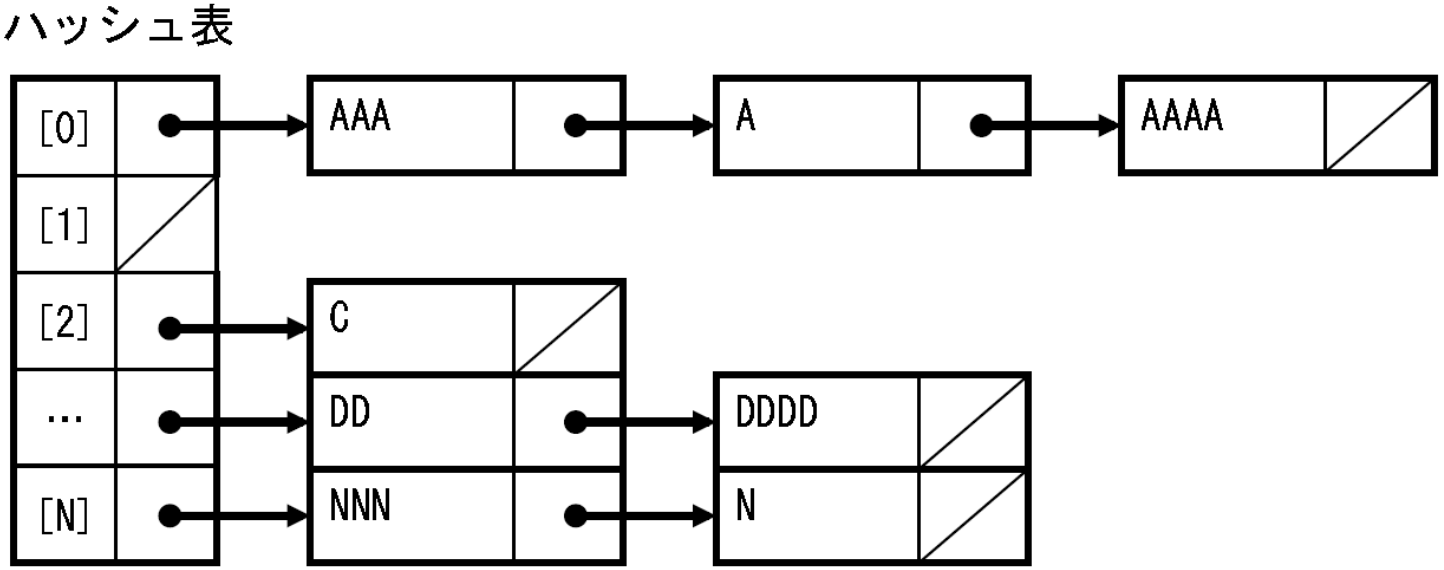
Chained hash (1)

- Chained hash
  - If a collision happens (a datum appears which has the same hash value), the datum will be stored into the same bucket by using linear list (singly linked list).



チェイン法によるハッシュ（1）

- チェイン法
  - 衝突が起きた（同じハッシュ値を持つものが現れた）ら、それらを線形リストで繋いで、同じバケットに保持する方法
  - ハッシュ値を計算したあと、線形リストを探索する



Chained hash (2)

- Insertion/deletion of a component is the same as for linear list after checking the hash table.

- Searching a linear list is not *O(1)*.
  - Checking the hash table is *O(1).*
  - Searching a linear list is *O(L)* if the length of the list is *L*.
    - Average length of the lists
      ＝The number of data／Hash table size
    - The chained hash search closes *O(1)* if the hash table size is enough big for the number of data and by using good hash function with less collisions.

チェイン法によるハッシュ（2）

- 要素の追加・削除は、ハッシュ表を調べた後は、線形リストに対する要素の追加・削除と同じ

- 線形リストの探索は*O(1)*じゃない
  - ハッシュ表を調べるところまでは*O(1)*
  - 線形リストの長さが *L* なら、探索は*O(L)*
    - リストの平均長
      ＝格納するデータ数／ハッシュ表の大きさ
    - データ数に対してハッシュ表が十分に大きく、衝突が頻繁におきないハッシュ関数を用いれば、*O(1)*に近づく