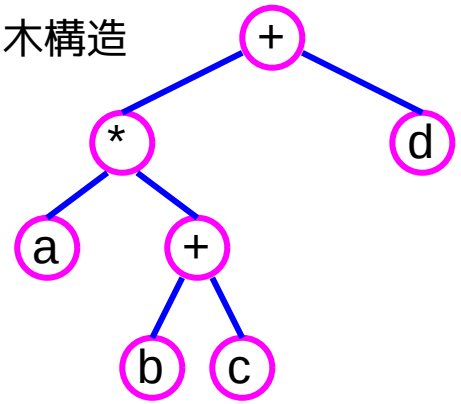
12 動的データ構造

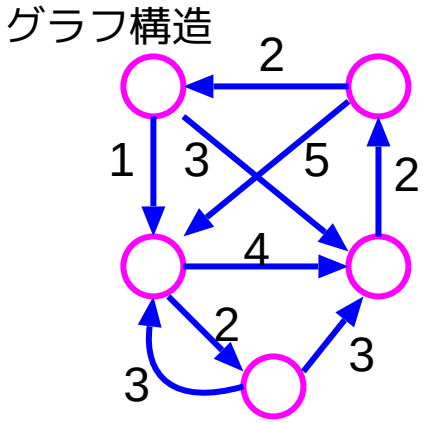
12–1 動的データ構造

配列と構造体だけを用いて構築されたデータ記憶領域は塊になっており、実行の途中にその形や大きさを変更することは(原則として)出来ない。
(⇨ 静的データ構造という。)

一方、小さめの記憶領域を動的に(i.e. 実行中に)確保し、それらをポインタでつなげば色々な形/大きさのデータ構造(i.e. データを記憶し操作するための表現形式)を表すことができ、また、実行中に(i.e. 動的に)形や大きさを変えることが出来る。 (⇨ 動的データ構造という。)

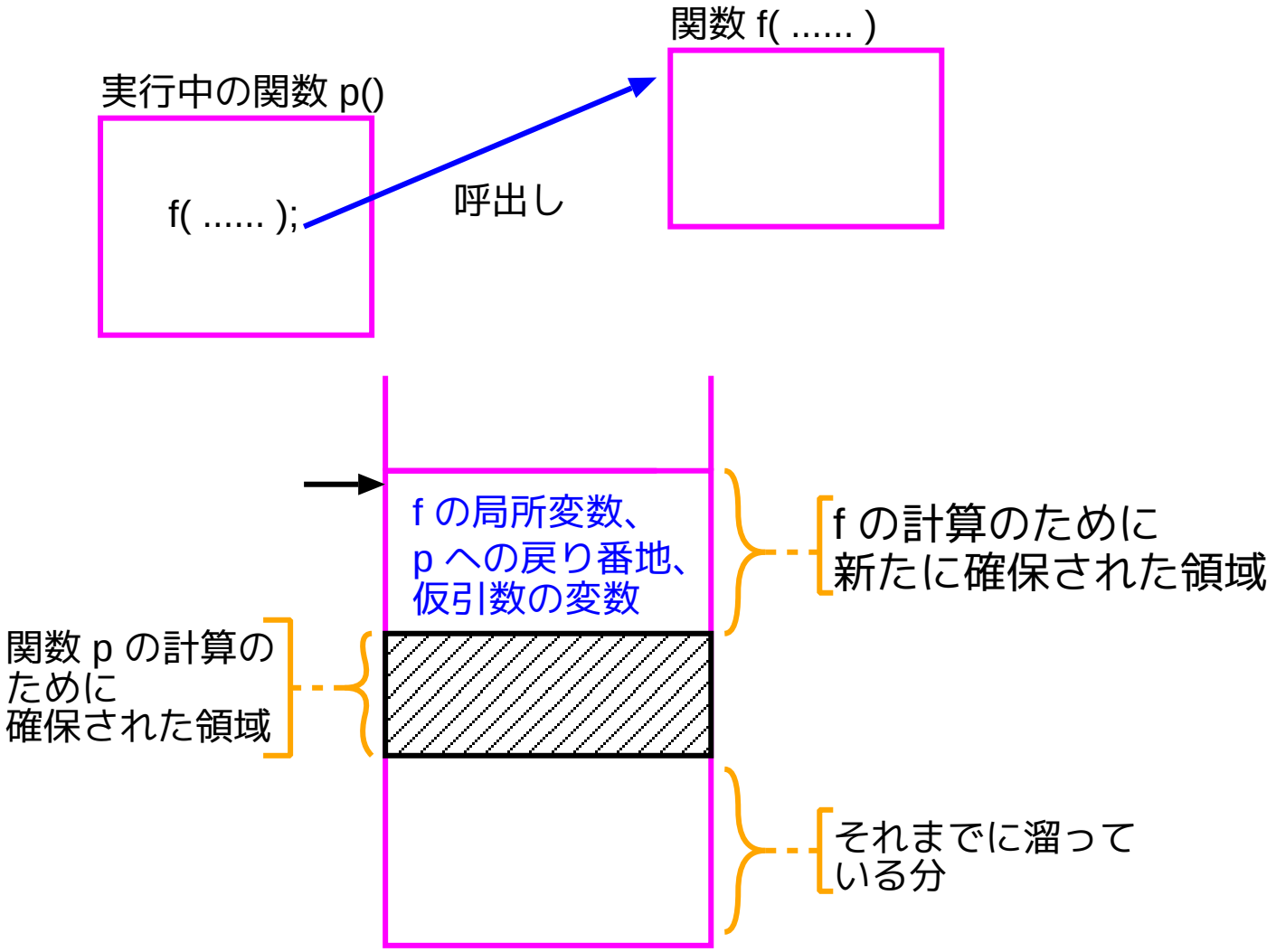
例12. 1





スタック領域とヒープ領域

講義ノート7.5節で述べた様に、関数の本体部(あるいはブロック)の最初に宣言された変数／配列(のためのメモリ)は、関数が呼ばれたり新しいブロックに入ったりする度にスタック領域上に確保される。



一方、計算機内部では、動的な記憶領域確保の要求に応えるための領域（ヒープ領域という）が用意されており、

- `malloc`や`calloc`といったライブラリ関数の呼び出しに対してこの中の小領域を割り当てたり、

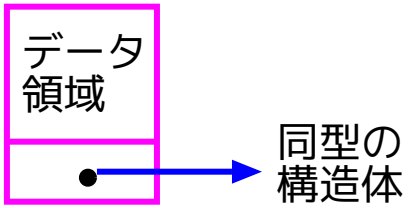
- `free`というライブラリ関数の呼び出しに対して不要になった小領域を再利用可能なメモリとして登録し直したり、

といった作業がなされる。

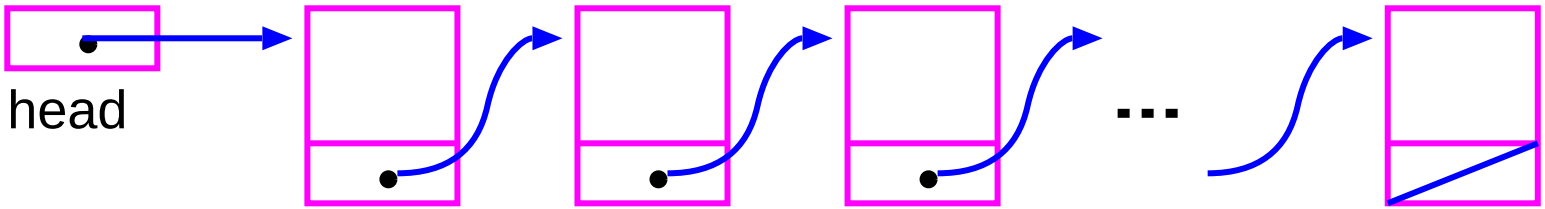
12–2 自己参照的構造体

自己参照的構造体： ···自分と同型の構造体へのポインタをメンバに持つ構造体

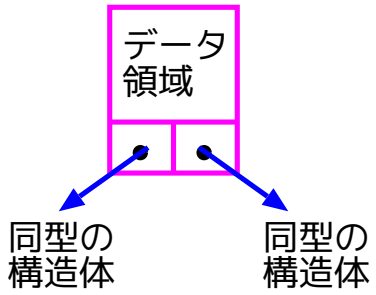
- 「自分と同型の構造体へのポインタ」を1個だけ持つ構造体は次のような構成をしている。



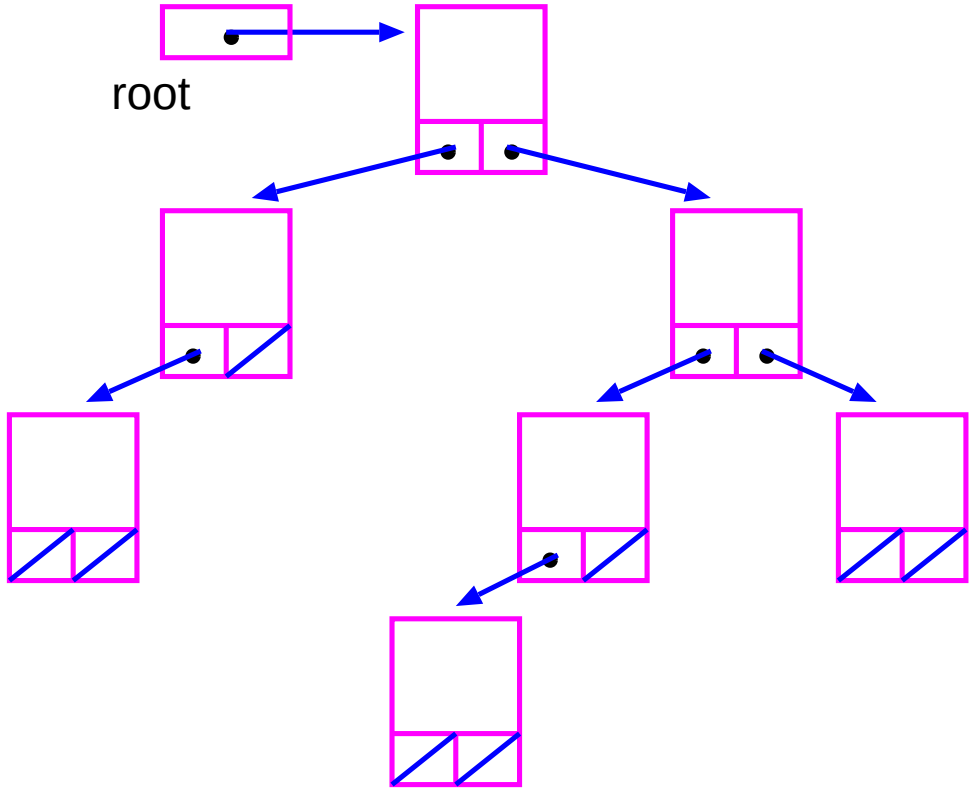
従って、この種の構造体からは次のような形の(動的)データ構造を構成することが出来る。このような構造を線形リストまたは連結リストという。



- 「自分と同型の構造体へのポインタ」を2個だけ持つ構造体は次のような構成をしている。



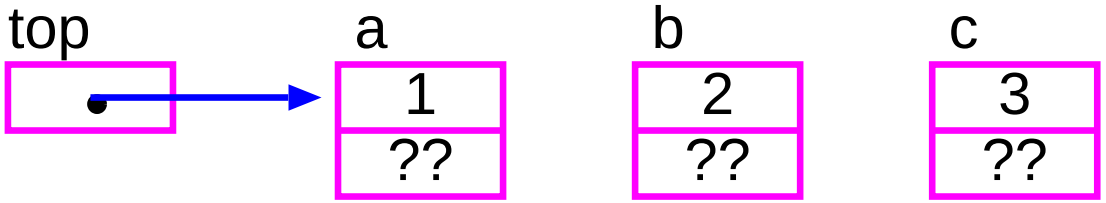
従って、この種の構造体からは次のような形の(動的)データ構造を構成することが出来る。このような構造を2分木という。



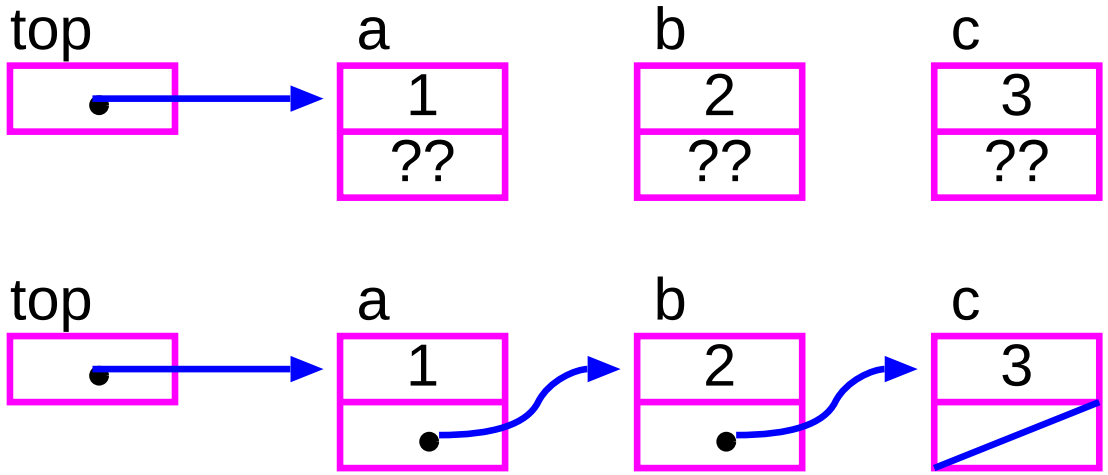
例12．2（自己参照的構造体）小さな線形リストを構成するCプログ...

[motoki@x205a]$ nl dynamic-build-small-llist.c

```
 1  #include <stdio.h>

 2  struct node {
 3    int           data;
 4    struct node  *next;    /* 自己参照的 */
 5  };

 6  int main(void)
 7  {
 8    struct node  *top, a, b, c;

 9    a.data = 1;
10    b.data = 2;
11    c.data = 3;

12    top    = &a;
```



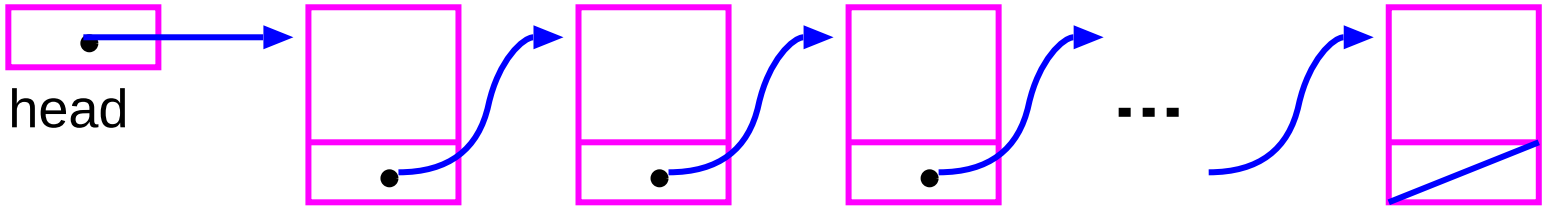
```
13      a.next = &b;
14      b.next = &c;
15      c.next = NULL;
```




```
16      printf("%3d%3d%3d\n", top->data,
                top->next->data,
17              top->next->next->data);
18               /* a,b,cという変数名を使わずに
                                  アクセスできる */
19      return 0;
20  }
[motoki@x205a]$  gcc dynamic-build-small-llist.c
[motoki@x205a]$  ./a.out
  1  2  3
[motoki@x205a]$
```

12–3 線形リスト

この節では線形リスト、すなわち次のような形のデータ構造の扱い方を例示する。



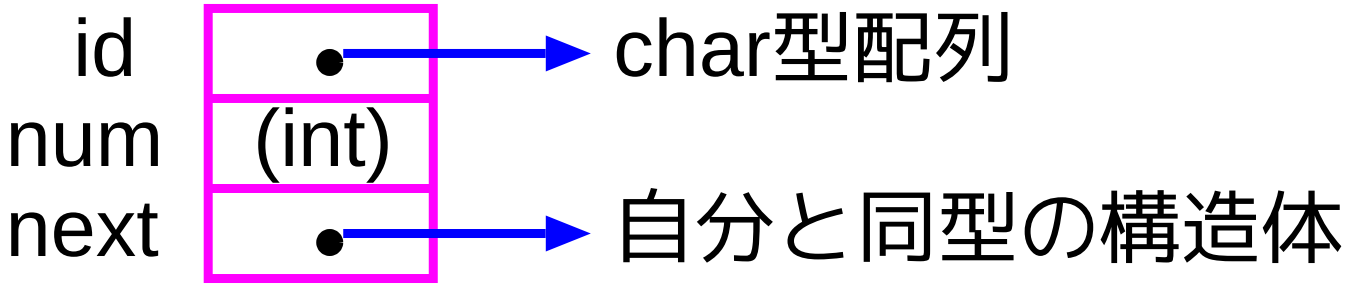
例題12．3（線形リスト）

① 入力ストリームに現れる 識別子 ＿＿ 整数 という形のデータを読み込んでは、
（＿＿：1個以上の空白）

② それを 識別子 に関して辞書順になる様に線形リストに登録する、

という作業を繰り返し、最後に線形リストに登録されたものを順に出力するプログラムを作成せよ。

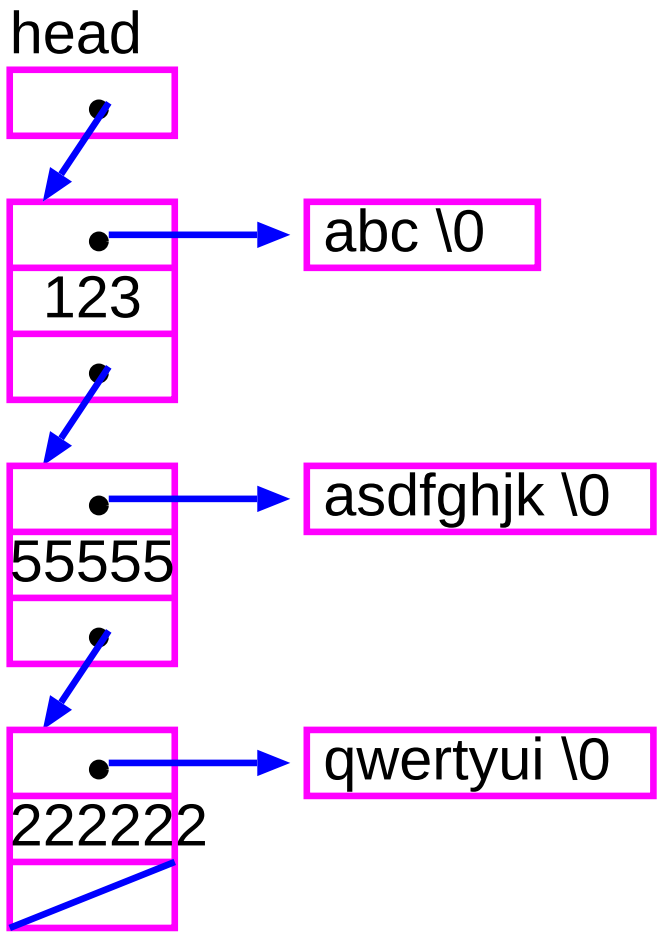
（考え方）　識別子 は英字で始まり英数字の続く文字列のことで、その長さに上限はない。従って、読み込んだ識別子全体を構造体の中に保持することは出来ない。そこで、線形リストを構成する自己参照的構造体として次の様なものを考える。



すると、例えば

```
asdfghjk 55555
qwertyui 222222
abc      123
```

という入力に対して



という風な線形リストを保持することにする。ここでさらに、

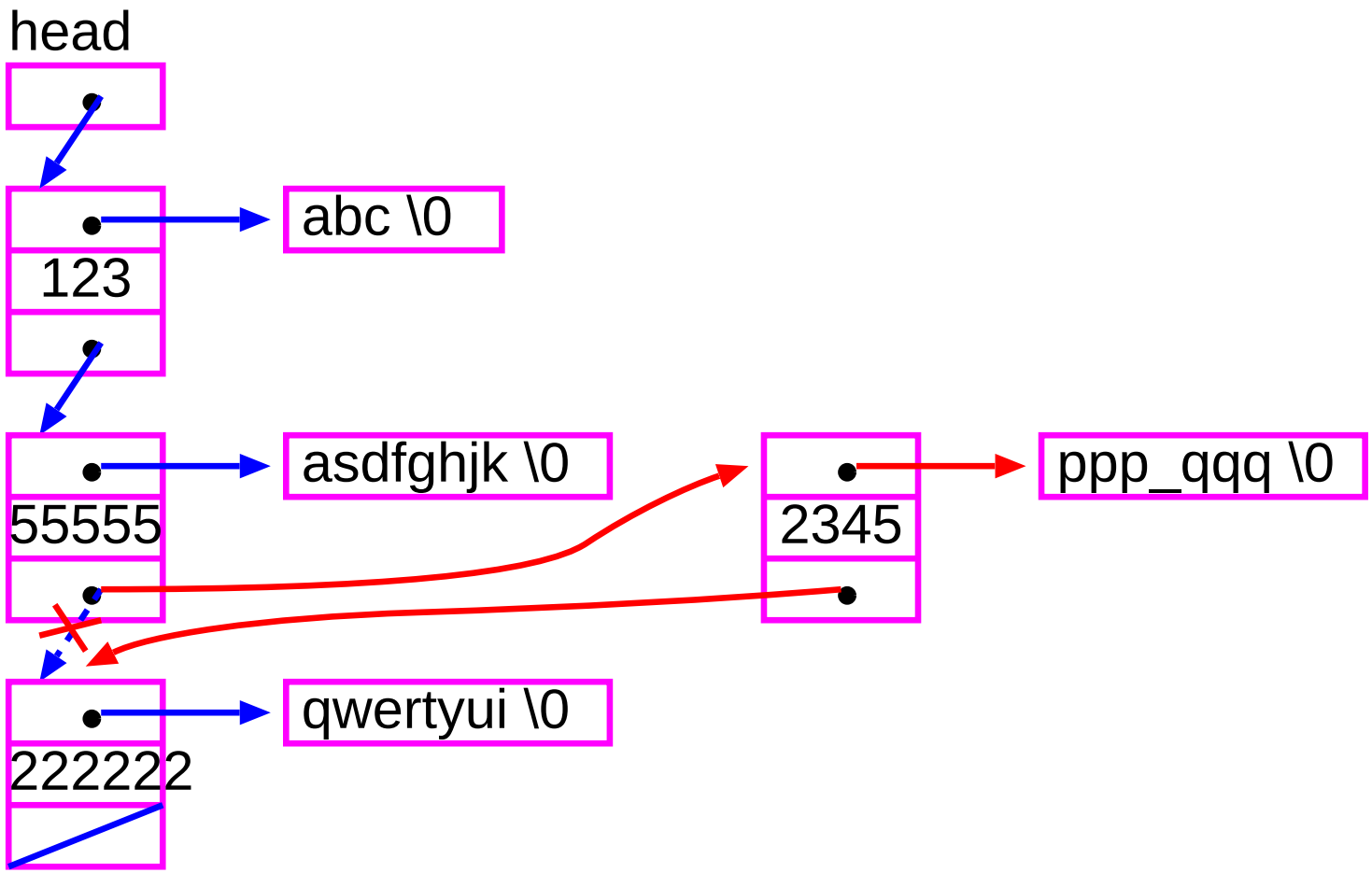
```
ppp_qqq  2345
```

という入力があれば、挿入位置の探査、必要なメモリの確保等を行なった上で次のようなポインタの付け替えを行なえばよい。

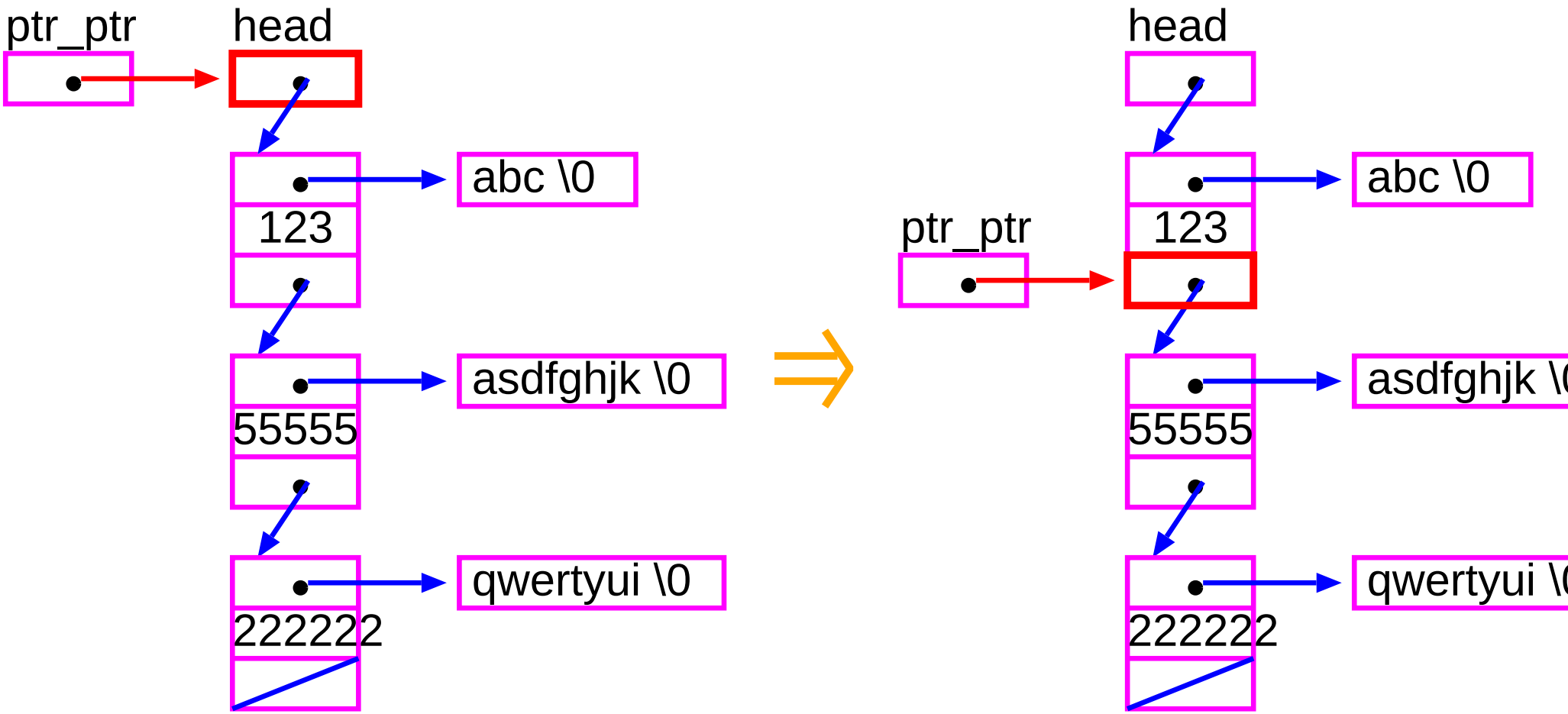
という風な線形リストを保持することにする。ここでさらに、

```
ppp_qqq  2345
```

という入力があれば、挿入位置の探査、必要なメモリの確保等を行なった上で次のようなポインタの付け替えを行なえばよい。



「挿入位置の探査」を行うには 、次の様に線形リストを先頭から順にたどり、各々の時点で、この次が新しい要素の挿入場所かどうかを標準ライブラリ関数 `strcmp()` を用いて判定すれば良い。



「必要なメモリの確保」を実行時に動的に行うには 、標準ライブラリ関数 `malloc( )` を用いれば良い。

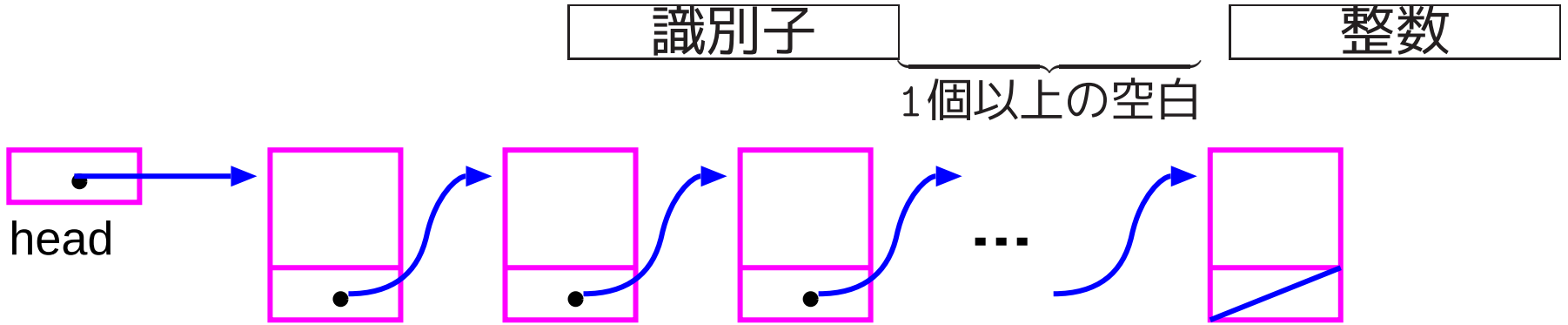
（プログラミング） プログラムの見通しを良くするために、入力した識別子と整数の組を（辞書順を保つ様に）線形リストに挿入する機能を果たす関数 `add_item( )` と、線形リストとして繋げられたデータを先頭から順に出力する機能を果たす関数`listprint( )` を用意した。

そして、簡単のため、入力したものが本当に識別子になっているかどうかのチェックは省略してプログラムを構成した。

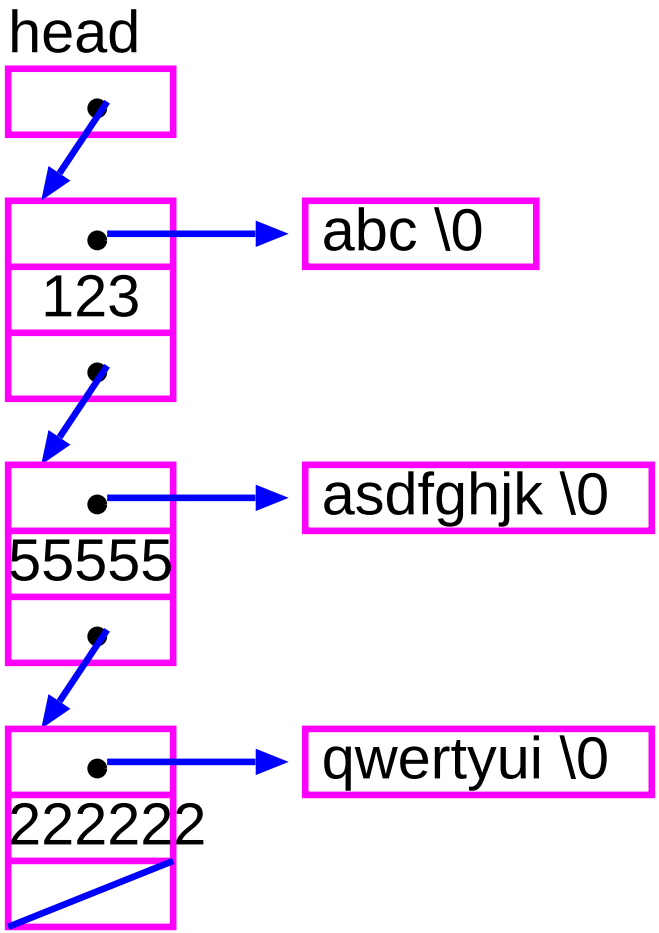
```
[motoki@x205a]$  nl dynamic-llist.c
     1 /*************************************************************
     2 /*
     3 /* 線形リストを用いて識別子(と整数)を辞書順に登録･出力 */
     4 /*
     5 /*************************************************************
     6
     7 #include <stdio.h>
     8 #include <string.h>
     9 #include <stdlib.h>
    10
    11 #define  MAXLENGTH  100
    12
    13 #define Is_empty(list)  ((list) == NULL)
    14
    15 typedef char *String;
    16
```

```
17 typedef  struct list_item  *List;
18 typedef struct list_item {
19     String id;
20     int    num;
21     List   next;
22 } List_item;
23
24 void add_item(List *ptr_ptr, String id, int num);
25 void listprint(List);
26
27 int main(void)
28 {
29   List head;
30   char identifier[MAXLENGTH+1];
31   int  count, num;
32
```

```
33    head = NULL;
34    while ((count=scanf("%100s %d",identifier,&num))==2)
35      add_item(&head, identifier, num);
36
37    if (count != EOF) {
38      printf("Input Error!\n");
39      exit(EXIT_FAILURE);
40    }
41
42    listprint(head);
43    return 0;
44  }
45
```



線形リストが



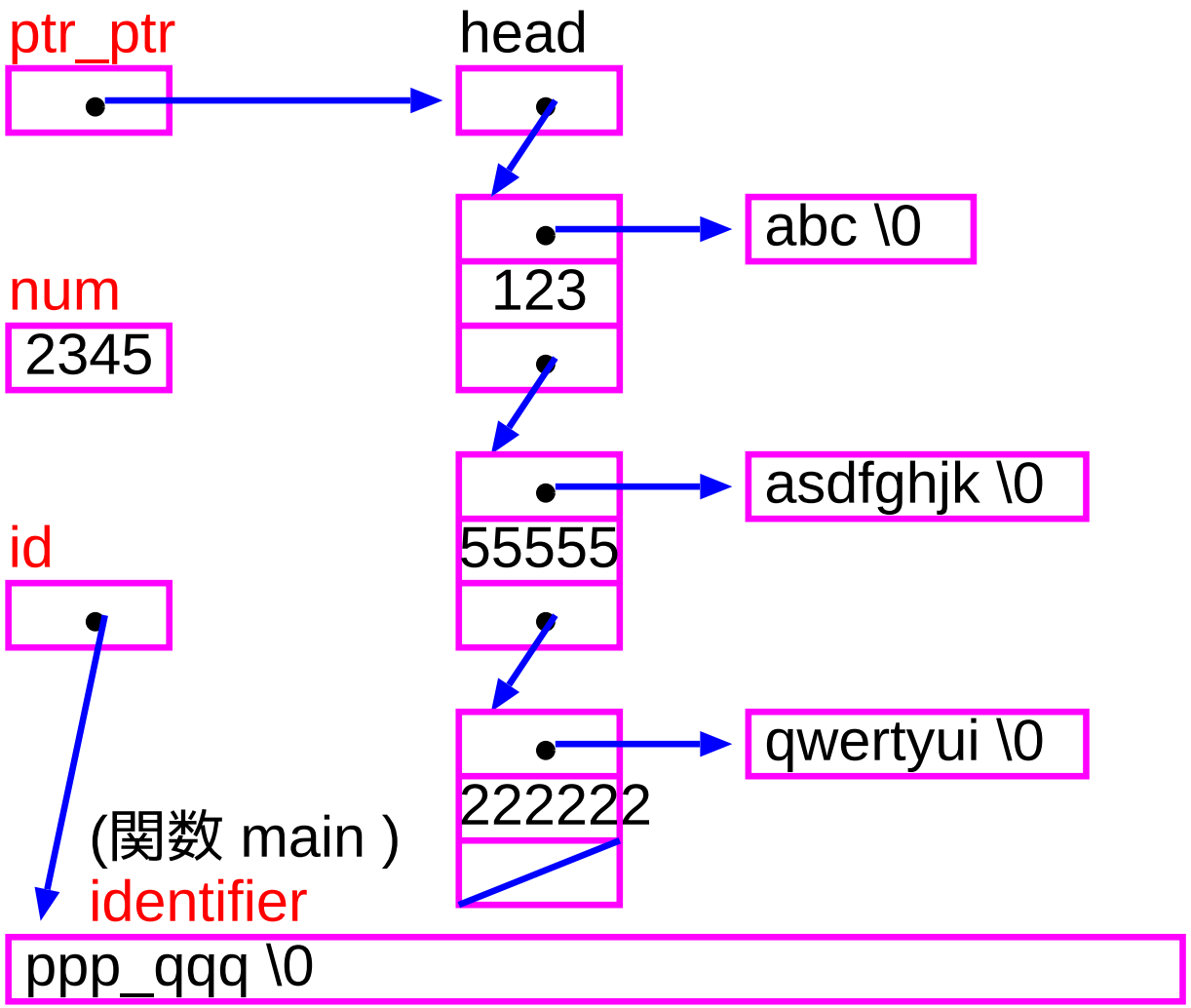
という状態の時、

```
ppp_qqq  2345
```

という入力を処理するために関数 `add_item` が呼び出されたとすると、...

```
ppp_qqq  2345
```

という入力を処理するために関数 `add_item` が呼び出されたとすると、
`add_item` の仮引数は次の様に設定される。



```
46 /*-----------------------------------------------
47 /* 線形リストへ識別子と整数の組を追加
48 /*-----------------------------------------------
49 /* (仮引数) ptr_ptr : "線形リストへのポインタ領域"への...*
50 /*            id      : 文字列データへのポインタ
51 /*            num     : 整数
52 /* (関数値) : なし
53 /* (機能)   : 識別子と整数の組を1つの項目として、それら...
54 /*            の辞書順に線形リストの形に繋げられており...
55 /*            *ptr_ptr がその線形リストの先頭を指し示す..
56 /*            る。この仮定の下で、id の指し示す識別子と...
57 /*            の組を新しい項目として(辞書順を保つ様に)...
58 /*            ストに挿入する。
59 /*-----------------------------------------------
60 void add_item(List *ptr_ptr, String id, int num)
61 {
62   List_item *new_item;
```

```
63
64    while (! Is_empty(*ptr_ptr)
                && strcmp(id, (*ptr_ptr)->id) > 0)
65      ptr_ptr = &((*ptr_ptr)->next);     /* 挿入場所を探す
66
67    new_item = (List_item *) malloc(sizeof(List_item));
68                                          /* 新しいitem枠を作る
69    if (new_item == NULL) {
70      printf("*** fail in memory allocation ***");
71      exit(EXIT_FAILURE);
72    }
73
74    new_item->id
        = (String) malloc(strlen(id)+1); /*idを入れる領域*/
75    if (new_item->id == NULL) {          /*を確保し、そこ*/
76      printf("*** fail in memory allocation ***");/*へ*/
77      exit(EXIT_FAILURE);                /*のポインタを代入*/
```

```
78    }
79    strcpy(new_item->id, id);      /* 新領域にidを複写 */
80
81    new_item->num = num;
82
83    new_item->next  = *ptr_ptr; /* ポインタの付け替え */
84    *ptr_ptr = new_item;
85 }
86
87 /*-------------------------------------------------
88 /* 線形リストの中味を出力
89 /*-------------------------------------------------
90 /* (仮引数) ptr : 線形リストへのポインタ
91 /* (関数値) : なし
92 /* (機能)   : 識別子と整数の組を1つの項目として、それら...
93 /*              の辞書順に線形リストの形に繋げられており、...
94 /*              その線形リストの先頭を指し示すと仮定する...
```

```
 95 /*                この仮定の下で、線形リストに記憶されたデー..
 96 /*                        番号    整数    識別子
 97 /*                という書式で出力する。
 98 /*-------------------------------------------------------
 99 void listprint(List ptr)
100 {
101   int n;
102
103   printf("番号          整数    識別子\n");
104   for (n=1; ! Is_empty(ptr); ptr = ptr->next, ++n)
105     printf("%4d%11d  %s\n", n, ptr->num, ptr->id);
106 }
[motoki@x205a]$  cat dynamic-llist.data
asdfghjk 55555
qwertyui 222222
abc      123
ppp_qqq  2345
```

```
zxcv      987
kkk       456
efghi     77
[motoki@x205a]$  gcc dynamic-llist.c
[motoki@x205a]$  ./a.out <dynamic-llist.data
番号          整数  識別子
   1           123  abc
   2         55555  asdfghjk
   3            77  efghi
   4           456  kkk
   5          2345  ppp_qqq
   6        222222  qwertyui
   7           987  zxcv
[motoki@x205a]$
```

配列 vs. 線形リスト

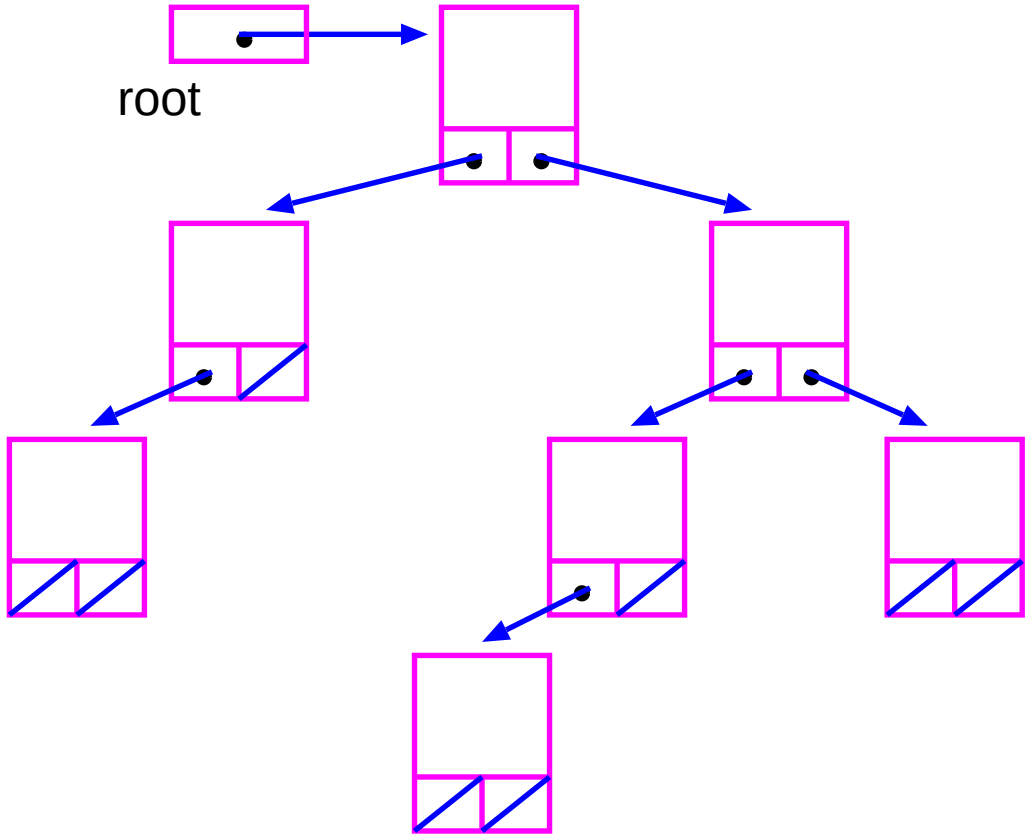
- どちらも、データを一列に並べたものを表せる。

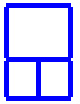
- 各々の特徴は次の通り。

配列	...	要素の挿入・削除がない場合は、 アクセス時間、記憶容量の点で最適な方法。
線形リスト	...	要素の挿入・削除が簡単に行なえる。 しかし、アクセスに時間がかかる。

12–4 2分木データ構造

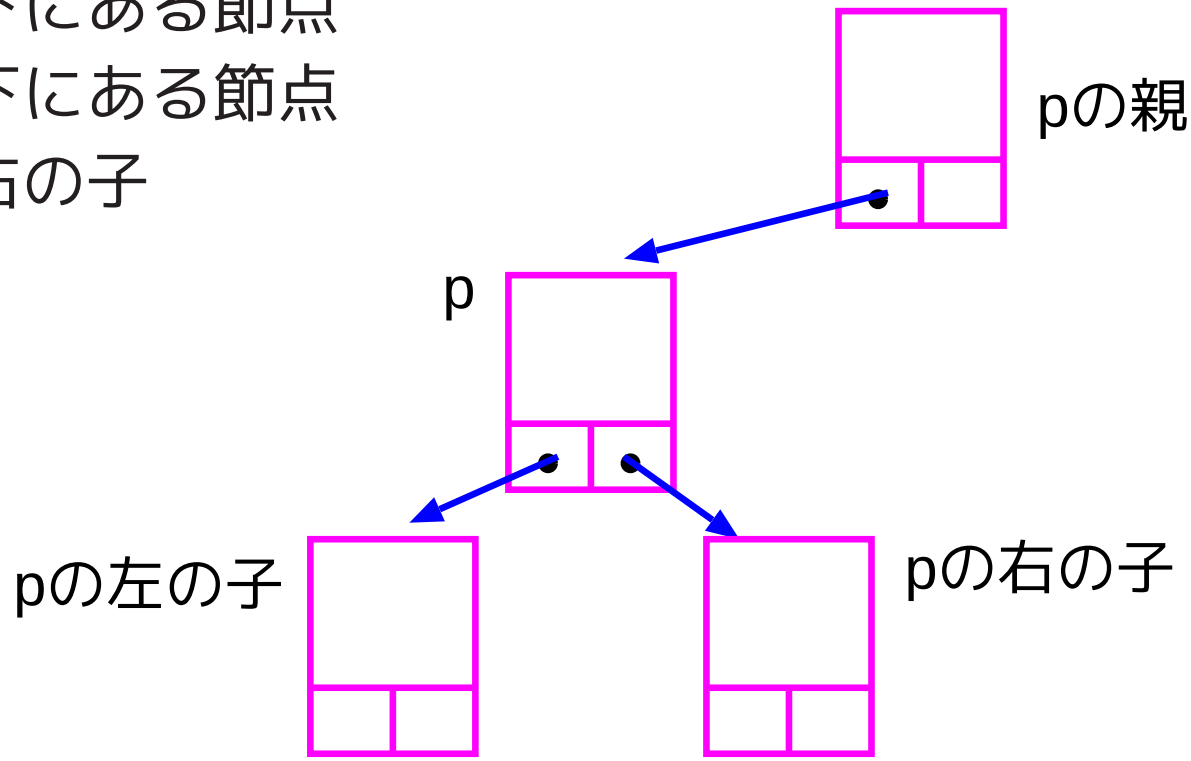
この節では2分木構造、すなわち次のような形のデータ構造の扱い方を例示する。



一般に、このような構造の構成要素である構造体  を節点と言い、節点と節点を結ぶ線を枝と言う。

2分木では、

p の親　　… p の上方にある節点
p の左の子 … p の左下にある節点
p の右の子 … p の右下にある節点
p の子　　… p の左右の子



根 … 親を持たない節点
葉 … 子を持たない節点
節点のレベル … 根とその節点を結ぶのに必要な枝の本数
2分木の高さ … レベルの最大値

例題12．4（2分木の中間順走査）　例題**12.3**では、①入力ストリームからデータを読み込んでは ②そのデータ内の 識別子 項目に関して辞書順になる様に線形リストに挿入してゆき、最後に線形リストに保存されたものを順に出力するプログラムを示した。配列ではなく線形リストを用いれば扱えるデータ数に上限がなくなるが、線形リストだと読み込んだデータを挿入する場所を探すのに相当の手間がかかってしまう。
そこで、常に

(左の子とその子孫の識別子) $\leq$ (親の識別子)

(親の識別子) $<$ (右の子とその子孫の識別子)

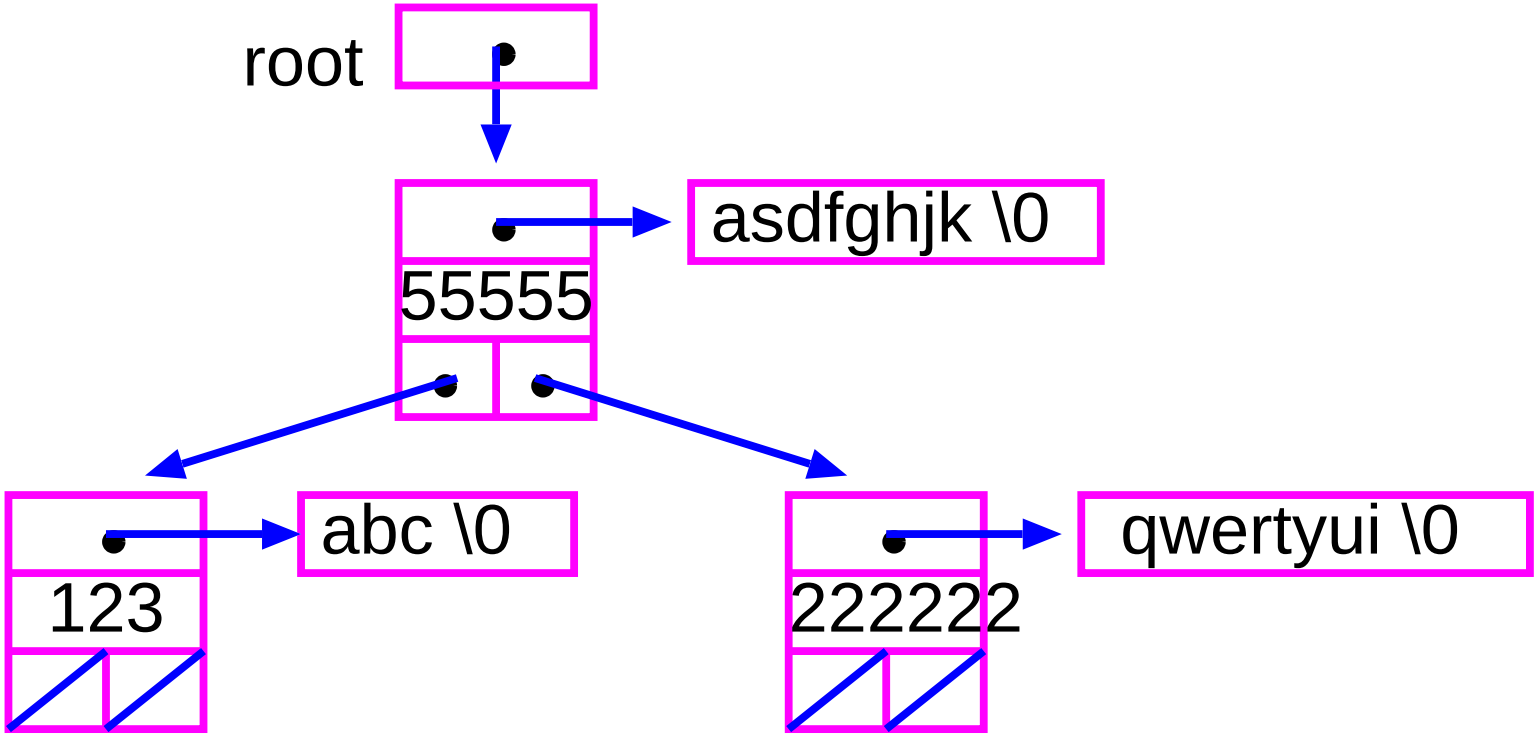
を満たすようにデータを記録すれば、データを辞書順に**2**分木上に保存できることに着目する。線形リストではなくこの様な**2**分木状に識別子と整数の組を保持することにして、例題**12.3**のプログラムを作り変えてみよ。

（考え方）　常に

（左の子とその子孫の識別子）$\leq$（親の識別子）

（親の識別子）$<$（右の子とその子孫の識別子）

を満たすようにデータを記録する訳だから、出来上がった2分木は例えば次の様なものになる。



求められているプログラムは 、

① 入力ストリームに現れる 識別子 ＿＿ 整数 という形のデータを読み込んでは、
（＿＿：1個以上の空白）

② それを、常に

（左の子とその子孫の識別子）≤（親の識別子）

（親の識別子）＜（右の子とその子孫の識別子）

を満たすように2分木に登録する、

という作業を繰り返し、最後に2分木に登録されたものを識別子の辞書順に出力することになる。

このプログラムを作成するに当たって考慮すべき、最も重要なことは

- 2分木内に蓄えられたデータをどの様な手順で辞書順に取り出すか。
- 入力した識別子と整数の組を（辞書順を保つ様に）2分木のどこに、またどの様な手順で挿入するか。

2分木内に蓄えられたデータを辞書順に全て取り出したい時、

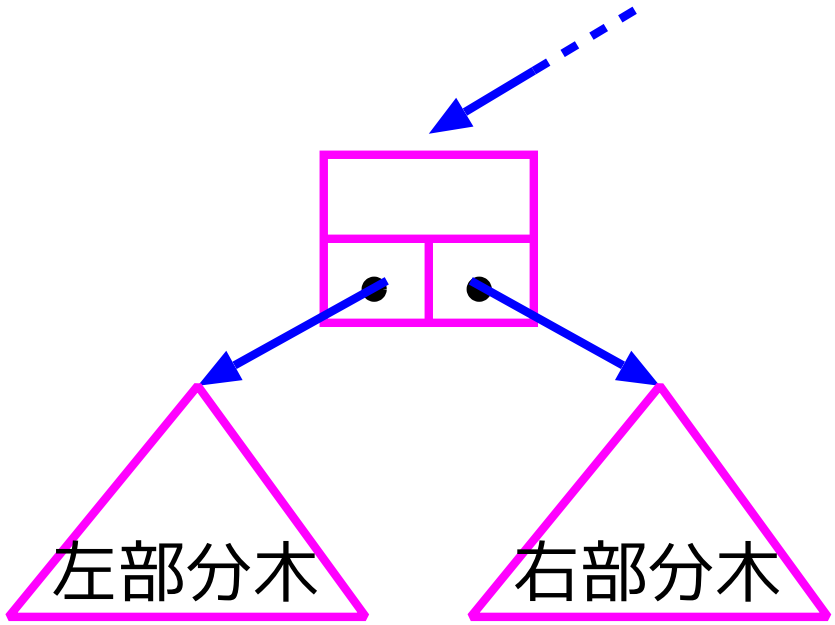
2分木の節点を再帰的にたどり、各々の節点で

① 左部分木内に蓄えられたデータを辞書順に全て取り出す（再帰）、

② 立ち寄った節点に蓄えられたデータを取り出す、

③ 右部分木内に蓄えられたデータを辞書順に全て取り出す（再帰）、

ということを行えば良い。



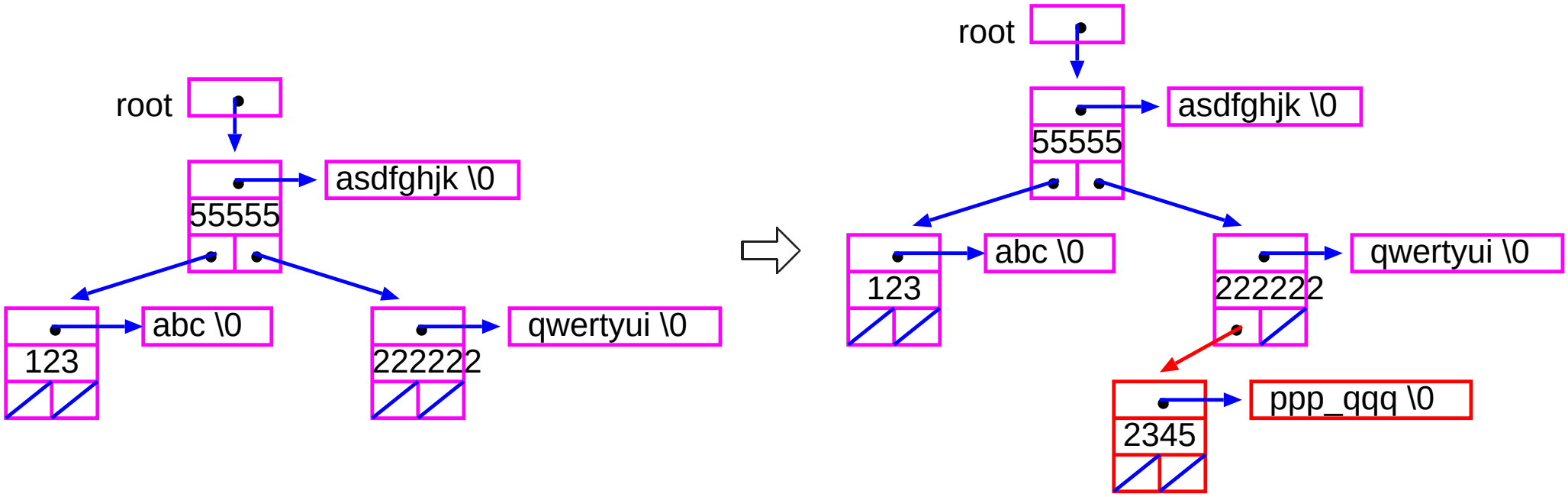
2分木に新しいデータを追加したい時、

2分木の葉の先の何処か然るべき場所を見つけてそこに新データを挿入することにより、辞書順を維持できる。

例えば、左の図のような2分木が出来ていた時、新しいデータとして

ppp_qqq　2345

というものがあれば、これを次の様に追加すれば良い。



そして、この追加作業は2分木の再帰的な構造に沿って行えば良い。

そして、この追加作業は2分木の再帰的な構造に沿って行えば良い。実際、ある与えられた節点 v 以下の2分木に新データを挿入したい場合は 、新データを v の左部分木に挿入すべきか、右部分木に挿入すべきかの判断をして、

もし左部分木に挿入すべきと判断されたら

①新データを v の左部分木に挿入(再帰)して

②出来上がった新しい左部分木へのポインタを

v の構造体の中に格納する。

そして、もし右部分木に挿入すべきと判断されたら

①新データを v の右部分木に挿入(再帰)して

②出来上がった新しい右部分木へのポインタを

v の構造体の中に格納する、

ということを行えば良い。

注目：

新しいデータを登録する際、例12.3では挿入場所を探す作業をwhile文による繰り返しで行ったが、ここでは2分木の構造に沿った再帰で行っている。

自己参照的構造体をつなげて出来るデータ構造は再帰的な構造をしているため、一般にこれらのデータ構造の処理は繰り返しでうまく書き表せるとは限らない。 この例のようにデータ構造の持っている再帰的な構造に沿ってプログラムを再帰的に構成する方が無難である。

（プログラミング）　プログラムの見通しを良くするために、入力した識別子と整数の組を（辞書順を保つ様に）2分木に挿入する関数 add_item( ) と、2分木内に蓄えられたデータを辞書順に出力し同時にメモリ解放も行う関数 inorder_traverse_and_print_free( ) を用意してプログラムを構成した。

```
[motoki@x205a]$  nl dynamic-btree.c
     1 /*************************************************************
     2 /*
     3 /*  2 分木上に識別子 (と整数) を辞書順に登録し、最後に 2 分...
     4 /*  中間順に走査することによって蓄えられたデータを辞書順に..
     5 /*
     6 /*************************************************************

     7 #include <stdio.h>
     8 #include <string.h>
     9 #include <stdlib.h>
```

```
10 #define  MAXLENGTH  100      /* 識別子の最大の長さ */

11 #define Is_empty(btree)  ((btree) == NULL)

12 typedef char *String;

13 typedef  struct btree_item  *Btree;
14 typedef struct btree_item {
15     String id;
16     int    num;
17     Btree  left_child, right_child;
18 } Btree_item;

19 Btree add_item(Btree tree, String id, int num);
20 void inorder_traverse_and_print_free(Btree);

21 int main(void)
```

```
22 {
23   Btree root;
24   char  identifier[MAXLENGTH+1];
25   int   count, num;

26   root = NULL;
27   while ((count=scanf("%100s %d",identifier,&num))==2)
28     root = add_item(root, identifier, num);

29   if (count != EOF) {
30     printf("Input Error!\n");
31     exit(EXIT_FAILURE);
32   }

33   printf("番号        整数   識別子\n");
34   inorder_traverse_and_print_free(root);
35   return 0;
```

```
36 }

37 /*--------------------------------------------------------
38 /*  2分木上に識別子と整数の組を追加
39 /*--------------------------------------------------------
40 /* (仮引数) tree : データの登録先となる2分木へのポイ...
41 /*             id   : 文字列データへのポインタ
42 /*             num  : 整数
43 /* (関数値) : データ登録後の2分木へのポインタ
44 /* (機能)   : 識別子と整数の組を1つの項目として、それら...
45 /*                  左の子とその子孫の識別子 <= 親の識別子
46 /*                  親の識別子 <  右の子とその子孫の識別子
47 /*               を常に満たすように2分木状に保存されており..
48 /*               tree がその2分木の根を指し示すと仮定する...
49 /*               この仮定の下で、id の指し示す識別子と整数 nu
50 /*               組を新しい項目として(辞書順を保つ様に)2...
51 /*               挿入して、出来た2分木へのポインタを返す。
```

```
52 /*-----------------------------------------------------
53 Btree add_item(Btree tree, String id, int num)
54 {
55   Btree_item *new_item;

56   if (Is_empty(tree)) { /*新しいitemを作りそこへの
                                ポインタを返す*/
57     new_item
         = (Btree_item *) malloc(sizeof(Btree_item));
58     if (new_item == NULL) {
59       printf("*** fail in memory allocation ***");
60       exit(EXIT_FAILURE);
61     }
62     new_item->id = (String) malloc(strlen(id)+1); /*id.
63     if (new_item->id == NULL) {                    /*域.
64       printf("*** fail in memory allocation ***");/*そ.
65       exit(EXIT_FAILURE);                          /*ン.
```

```
66      }
67      strcpy(new_item->id, id);       /* 新領域にidを複写 */
68      new_item->num          = num;
69      new_item->left_child   = NULL;
70      new_item->right_child = NULL;
71      return new_item;
72    }else if (strcmp(id, tree->id) <= 0) { /* 左部分木へ.
73      tree->left_child
          = add_item(tree->left_child, id, num);
74      return tree;
75    }else {                                 /* 右部分木へ.
76      tree->right_child
          = add_item(tree->right_child, id, num);
77      return tree;
78    }
79  }
```

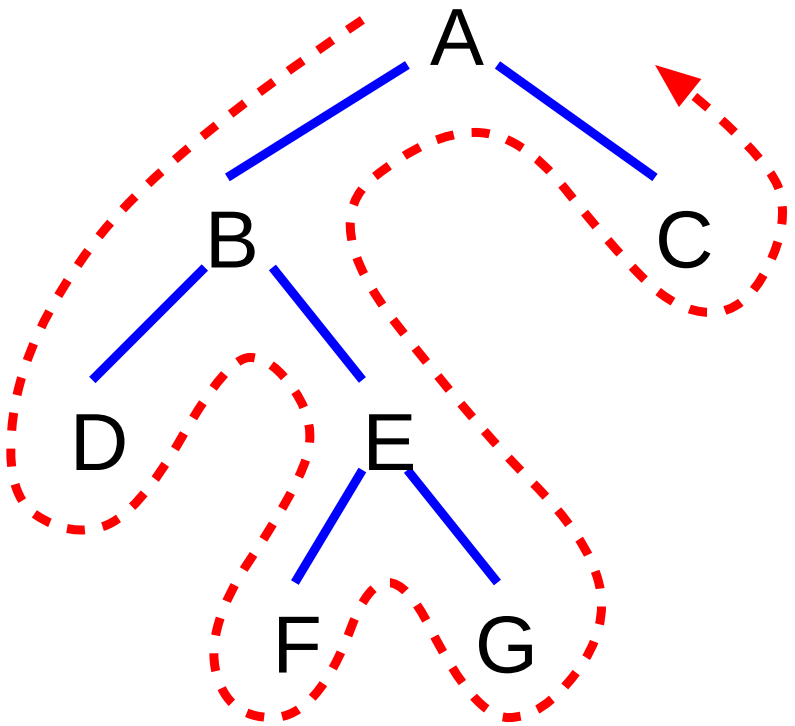
```
80 /*-----------------------------------------------
81 /*  2分木の中味を辞書順に出力，同時に2分木のメモリを解放
82 /*-----------------------------------------------
83 /* (仮引数) ptr :  2分木へのポインタ
84 /* (関数値) : なし
85 /* (機能)   : 識別子と整数の組を1つの項目として、それら...
86 /*                左の子とその子孫の識別子 <= 親の識別子
87 /*                親の識別子 <  右の子とその子孫の識別子
88 /*            を常に満たすように2分木状に保存されており..
89 /*            がその2分木の根を指し示すと仮定する。この..
90 /*            下で、2分木を中間順に走査することによって...
91 /*            木に蓄えられたデータを
92 /*                  番号   整数   識別子
93 /*            という書式で出力する。
94 /*            また、同時に2分木のメモリを解放する。
95 /*-----------------------------------------------
96 void inorder_traverse_and_print_free(Btree ptr)
```

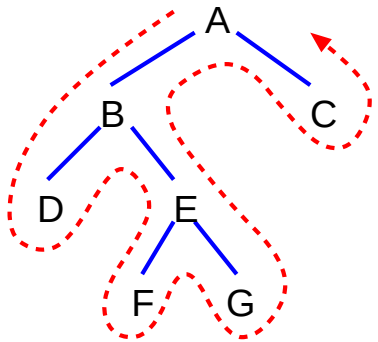
```
 97 {
 98   static int n=1;

 99   if (ptr != NULL) {
100     inorder_traverse_and_print_free(ptr->left_child);
101     printf("%4d%11d  %s\n", n++, ptr->num, ptr->id);
102     inorder_traverse_and_print_free(ptr->right_child);
103     free(ptr->id);
104     free(ptr);
105   }
106 }
[motoki@x205a]$  cat dynamic-llist.data
asdfghjk 55555
qwertyui 222222
abc      123
ppp_qqq  2345
zxcv     987
```

```
kkk         456
efghi       77
[motoki@x205a]$  gcc dynamic-btree.c
[motoki@x205a]$  ./a.out <dynamic-llist.data
番号          整数  識別子
   1           123  abc
   2         55555  asdfghjk
   3            77  efghi
   4           456  kkk
   5          2345  ppp_qqq
   6        222222  qwertyui
   7           987  zxcv
[motoki@x205a]$
```

2分木の全ての節点をたどる方法：

- 親から子へのポインタ（だけ）で節点を繋げて２分木を表す場合は、「深さ優先」でたどるしかない。





- 立ち寄った節点で何らかの処理を行いたい場合、
  木の構造に基づいて再帰的にアルゴリズムを書くことが出来る。

  左部分木の方を右部分木より先にたどることにすると、各節点の処理の順番としては次の３種類が可能。

  - 先行順…立ち寄った節点，左部分木，右部分木の順
  - 中間順…左部分木，立ち寄った節点，右部分木の順
  - 後行順…左部分木，右部分木，立ち寄った節点の順