# 7　データ構造

本節では総まとめとしてデータ構造の話を取り上げる。取り上げる話題としては，リスト構造，キュー構造，スタック構造，２分木ツリー構造である。

すでに一次元配列，二次元配列，構造体といった固定サイズのデータ構造については既に学んだ。本節では，データサイズが伸縮するようなデータ構造を取り上げる。本説で学ぶことは，今まで学習してきた要素がすべて入っているという意味で総まとめになっている。

## 7.1　自己参照構造体

自己参照構造体とは，自分自身へのポインタを含んだ構造体である。例えば次のような struct node という構造体の宣言を考える。

```
struct node{
  int data;
  struct node *nextPtr;
};
```

構造体 node は二つのメンバから成り立っている。一つは int 型のデータ data であり，もう一つは構造体 struct node へのポインタ, すなわち自分自身と同じ構造を持つデータへのポインタ nextPtr である。このような構造体のことを自己参照構造体という。

自己参照構造体を使って，nextPtr によってデータを結びつければ，リスト構造，スタック構造，リスト構造，二分木リストなど，さまざまなデータ構造を実現することができる。このようなデータ構造は，データサイズがあらかじめ分かっていないデータに対して当てはめるのが効果的である。あらかじめデータサイズが分かっていれば，配列を使うこともできるが，データサイズが，あらかじめ分かっていない場合には **動的データ構造** を使うことになる。

動的データ構造とは，malloc() を使った動的メモリ配置を使ったメモリ領域の確保の仕方をさす。すなわち，プログラムの実行時に，必要となるデータ領域を確保し (malloc())，不要になったら解放する (free()) 機構が必要である。ユーザがどれほどのメモリ領域を使うことができるかは，コンピュータのその時の状態による。ユーザのために利用できるメモリ領域がたくさん残っていれば，より多くのメモリを使うことができる。

動的メモリ配置には，malloc(), free(), および sizeof 演算子が不可欠である。malloc() 関数は，動的に確保するメモリのバイト数を引数にとり，void * へのポインタを返す。例えば以下の文

```
newPtr = (struct node *)malloc(sizeof(struct node));
```

は，sizeof (struct node) を評価して得られるバイト数に等しいメモリ領域を確保し，そのメモリ領域へのポインタを返す。それを (struct node *) でキャストして newPtr へ代入している。

動的に割り当てるべきメモリの確保に失敗すると malloc() は NULL ポインタを返す。従ってプログラマはメモリの割り当てに成功したか，失敗したかを malloc() が呼び出されるたびにチェックしなければならない。

free() は，malloc() によって確保されたメモリ領域を解放するために使われる。上述の malloc() で確保したメモリ領域を解放するためには,

```
free(newPtr);
```

とすればよい。このとき newPtr そのものは消去されず，newPtr の指しているメモリ領域が解放されるという点に注意。malloc() によって確保されたメモリ領域は，不要になったらすみやかに free() によって解放すべきである。そうしないと，システムが誤ってメモリ不足であると判断してしまうことになる。また，解放されたメモリ領域は参照すると実行時にエラーとなる場合があるので参照しないこと。

## 7.2 連結リスト

連結リストとは，ノードと呼ばれる自己参照構造体をリンクポインタで直線状に連結したものである。連結リストは，そのリストの最初のノードへのポインタを介してアクセスされる。それ以降のノードは，各ノードにあるポインタを順にたぐって行くことによって読み出せる。連結リストの最終ノードのリンクポインタには NULL がセットしてあってリストの終わりであることを知ることができる。

データのリストは配列にも格納できるが，連結リストの方がいくつか利点がある。連結リストは格納するデータサイズがあらかじめ予測できない場合に適している。連結リストは動的なデータ構造であるので，データの長さは必要に応じて伸縮することできる。一方，配列のサイズはコンパイル時に固定されてしまうので変更することができない。

連結リストでは，新しい要素をリストの中の適切な位置に挿入するだけでデータの順序をソートされた状態に保つことができる。一方，連結リストの欠点としては，データを挿入したり，削除したりするときに，絶えずそのリスト構造を参照しなければならないため，操作のオーバーヘッドが発生する。配列の要素は，メモリ内に連続して格納される。そのため，どの要素のアドレスも配列の先頭からの相対位置 (オフセット) に基づいて直接計算できるので，任意の配列要素にダイレクトにアクセスできる。一方，連結リストでは配列の要素のように各要素を直接アクセすることができない。

実行時に伸びたり縮んだりするデータ構造に対して，配列の代わりに動的メモリ配置を使えば，メモリを節約することができるが，動的メモリ配置では，ポインタの分だけメモリを余分に消費すること，および，動的メモリ配置は関数呼び出しのオーバーヘッドを伴うことは覚えておかなければならない。

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  typedef struct list {
 5      int value;
 6      struct list *nextPtr;
 7  } LIST;
 8
 9  LIST *insert_list(LIST *list, int data);
10  LIST *delete_list(LIST *list, int data);
11  void print_list(LIST *list);
12  void print_num(LIST *list);
13  int isEmpty(LIST *list);
14
15  void usage(void)
16  {
```

```
17          printf("使い方: 1: リストにある数を追加\n");
18          printf("        2: リストからある数を削除\n");
19          printf("        3: リストの印刷\n");
20          printf("        4: リストの要素の数を数える\n");
21          printf("        5: 終了\n");
22      }
23
24      int main(void)
25      {
26          LIST *start = NULL;
27          int data;
28          int menu;
29
30          usage();
31          printf("メニュー番号を入力してください:");
32          scanf("%d", &menu);
33
34          while(menu != 5) {
35          switch(menu) {
36          case 1:
37              printf("リストに挿入する数を入力してください:");
38              scanf("%d", &data);
39              start = insert_list(start, data);
40              break;
41          case 2:
42              if ( isEmpty(start) ){
43              printf("リストは空です\n");
44              } else {
45              printf("削除する数を入力してください:");
46              scanf("%d", &data);
47              start = delete_list(start, data);
48              }
49              break;
50          case 3:
51              print_list(start);
52              break;
53          case 4:
54              print_num(start);
55              break;
56          default:
57              printf("番号が不正です\n");
58              usage();
59              break;
60          }
61          printf("メニュー番号を入力してください:");
```

```
62          scanf("%d", &menu);
63          }
64
65          return 0;
66      }
67
68      int isEmpty(LIST *list)
69      {
70          return list == NULL;
71      }
72
73      LIST *insert_list(LIST *listPtr, int data)
74      {
75          LIST *newPtr, *previousPtr, *currentPtr;
76
77          newPtr = (LIST *)malloc(sizeof(LIST));
78
79          if ( newPtr == NULL ) {
80          printf("メモリ不足でデータが割り当てられません\n");
81          } else {
82          newPtr->value = data;
83          newPtr->nextPtr = NULL;
84
85          previousPtr = NULL;
86          currentPtr = listPtr;
87
88          while ( currentPtr != NULL && data > currentPtr->value ) {
89              previousPtr = currentPtr;
90              currentPtr = currentPtr->nextPtr;
91          }
92
93          if ( previousPtr == NULL ) {
94              newPtr->nextPtr = listPtr;
95              listPtr = newPtr;
96          } else {
97              previousPtr->nextPtr = newPtr;
98              newPtr->nextPtr = currentPtr;
99          }
100         }
101         return listPtr;
102     }
103
104     LIST *delete_list(LIST *listPtr, int data)
105     {
106         LIST *previous, *current, *tmp;
```

```
107
108         if ( data == listPtr->value ) {
109             tmp = listPtr;
110             listPtr = listPtr->nextPtr;
111             free(tmp);
112         } else {
113             previous = listPtr;
114             current = listPtr->nextPtr;
115             while ( current != NULL && current->value != data ) {
116                 previous = current;
117                 current = current->nextPtr;
118             }
119             if ( current != NULL ) {
120                 tmp = current;
121                 previous->nextPtr = current->nextPtr;
122                 free(tmp);
123             }
124         }
125         return listPtr;
126     }
127
128     void print_list(LIST *listPtr)
129     {
130         if (listPtr == NULL) {
131         printf("リストは空です\n");
132         } else {
133         printf("リストの内容\n");
134         while ( listPtr != NULL) {
135             printf("%d ->", listPtr->value);
136             listPtr = listPtr->nextPtr;
137         }
138         printf("NULL\n\n");
139         }
140     }
141
142     void print_num(LIST *listPtr)
143     {
144         int n = 0;
145         while ( listPtr != NULL) {
146         n++;
147         listPtr = listPtr->nextPtr;
148         }
149         printf("要素の数は %d 個です\n", n);
150     }
```

連結リストの操作には，二つの関数 insert_list(), delete_list() を用いる。isEmpty() はリストが空であるか否かを判断する関数である。リストが空のとき 1 を返し，それ以外の時は 0 を返す。print_list() は連結リストの内容を表示し，print_num() は，連結リストの要素数を返す関数である。

データは，昇順にリストに挿入される。insert_list() は引数として連結リストのアドレスと挿入するデータを受け取って連結リストを返す関数である (ソースコード 73 行目より)

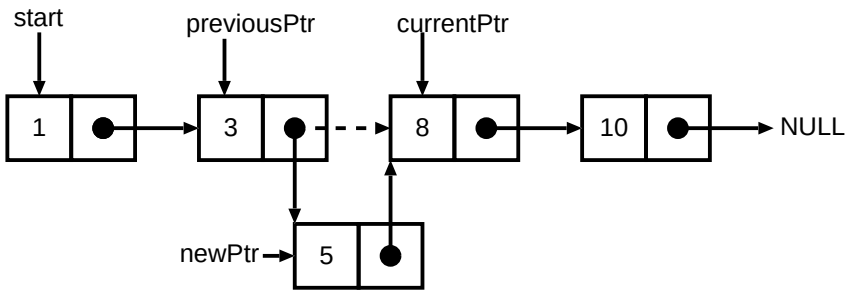


図 2: リストにノードを追加する insert_list() の仕組み

insert_list() (図 2 を参照) は,

1. malloc() を呼び出して，構造体 LIST のバイト数からなるメモリの領域を確保し，newPtr に代入し (77 行目)
2. 新しいデータで newPtr->value を初期化し (82 行目)
3. 新しいデータの nextPtr を NULL で初期化し (83 行目)
4. previousPtr を NULL で初期化し (85 行目)
5. currentPtr に現在の listPtr の値を代入している (86 行目)。previousPtr と currentPtr とはそれぞれ挿入する場所の直前のノードと直後のノードを指すために使われる。
6. currentPtr が NULL ではなく，かつ，data が currentPtr の value より大きい間は，currentPtr を previousPtr に代入し，currentPtr を次のノードに進める (88 行目から 90 行目)。この操作によって，データの挿入ポイントが決まる。
7. previousPtr が NULL の場合は，リストの先頭に新しいノードを挿入する (93 行目から 95 行目)
8. prevoiusPtr が NULL でない場合は，リストの途中に新しいノードを挿入する。newPtr を previousPtr の nextPtr に代入 (97 行目) することで，挿入ポイントの直前のノードの次のポインタが新しいノードを指すようにし，newPtr の nextPtr に currentPtr を代入することで (98 行目)，新しいノードが直後のノードを指すようにしている

図 3 は，delete_list() の動作を表している。

1. 削除するデータがリストの先頭にある場合には，listPtr の値（アドレス）を tmp に代入し (109 行目)，listPtr の nextPtr を listPtr の値としている (110 行目)。そして tmp の指す領域を free() で解放し (110 行目) ている。

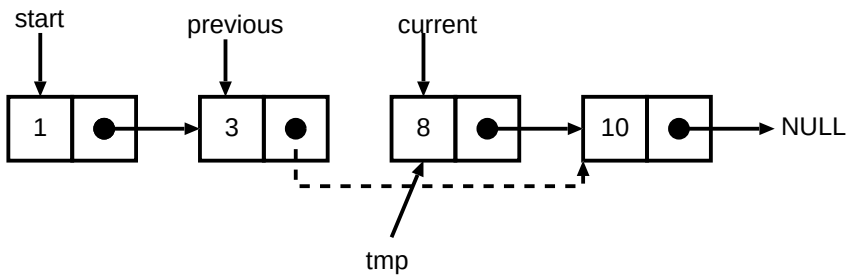


図 3: リストからノードを削除する delete_list() の仕組み

2. 削除するデータが先頭要素以外の場合には，previous を listPtr で初期化し (113 行目)，current を listPtr->nextPtr で初期化 (114 行目) してから，

3. previous を current で置き換え，current の nextPtr で current を置き換えることによって次のノードに進めている。

4. current が NULL でなければ，current を tmp に代入し (120 行目)，previous->nextPtr の値を current->nextPtr の値で置き換えている (121 行目)。その上で，tmp の値を free() で解放している。

**演習 7.1** 再帰を使って，リストを逆順に印刷する関数 printListBackward() を作れ

## 7.3 スタック構造

スタックとは制約付きリスト構造である。新しいノードは先頭ノードへしか追加できず，先頭からしか削除できない。このためスタック構造は Last-In First-Out (LIFO) 構造と呼ばれる。スタックの最終ノードのリンクメンバには，スタックの底を示すための NULL がセットされる。

スタックを操作するために使う関数は push() と pop() である。push() は新しいノードをスタックの先頭に挿入する。pop() は，スタックの先頭ノードからデータを取り除き，その値を返す。

以下のプログラムは，簡単なスタックを実装したものである。プログラムは 4 つの選択肢を選択することから始まる。

**1.** スタックに値をプッシュする

**2.** スタックから値をポップする

**3.** スタックを印刷する

**4.** 終了

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct stack {
5      int data;
```

```
 6      struct stack *nextPtr;
 7  } STACK;
 8
 9  STACK *push(STACK *stackPtr, int data);
10  STACK *pop(STACK *stackPtr, int *data);
11  int isEmpty(STACK *stackPtr);
12  void print_stack(STACK *stackPtr);
13  void usage(void);
14
15  void usage(void)
16  {
17      printf("使い方 1: スタックに値を push する\n");
18      printf("      2: スタックから値を pop する\n");
19      printf("      3: スタックを印刷する\n");
20      printf("      4: 終了\n");
21  }
22
23  int main(void)
24  {
25      STACK *stackPtr = NULL;
26      int data;
27      int menu;
28
29      usage();
30      printf("メニュー番号を入力してください:");
31      scanf("%d", &menu);
32
33      while(menu != 4) {
34      switch(menu) {
35      case 1:
36          printf("スタックに push する整数を入力してください:");
37          scanf("%d", &data);
38          stackPtr = push(stackPtr, data);
39          print_stack(stackPtr);
40          break;
41      case 2:
42          if ( !isEmpty(stackPtr) ) {
43              stackPtr = pop(stackPtr, &data);
44              printf("pop した値は %d です\n", data);
45          } else {
46              printf("スタックは空です\n");
47          }
48          break;
49      case 3:
50          if ( !isEmpty(stackPtr) )
```

```
51              print_stack(stackPtr);
52          else
53              printf("スタックは空です\n");
54          break;
55      default:
56          printf("番号が不正です\n");
57          usage();
58          break;
59      }
60      printf("メニュー番号を入力してください:");
61      scanf("%d", &menu);
62      }
63
64      return 0;
65  }
66
67  int isEmpty(STACK *stackPtr)
68  {
69      return stackPtr == NULL;
70  }
71
72  STACK *push(STACK *topPtr, int data)
73  {
74      STACK *newPtr;
75
76      newPtr = (STACK *)malloc(sizeof(STACK));
77
78      if ( newPtr != NULL ) {
79          newPtr->data = data;
80          newPtr->nextPtr = topPtr;
81          topPtr = newPtr;
82      } else {
83          printf("メモリが足りません\n");
84      }
85      return topPtr;
86  }
87
88  STACK *pop(STACK *topPtr, int *pop_value)
89  {
90      STACK *tmpPtr;
91
92      tmpPtr = topPtr;
93      *pop_value = topPtr->data;
94      topPtr = topPtr->nextPtr;
95      free(tmpPtr);
```

```
 96      return topPtr;
 97  }
 98
 99  void print_stack(STACK *stackPtr)
100  {
101      if (stackPtr == NULL) {
102          printf("スタックは空です\n");
103      } else {
104          while ( stackPtr != NULL ) {
105              printf("%d-> ", stackPtr->data);
106              stackPtr = stackPtr->nextPtr;
107          }
108          printf("NULL\n\n");
109      }
110  }
```

push() がスタックに新しいノードをプッシュするときの手順は以下のとおり。

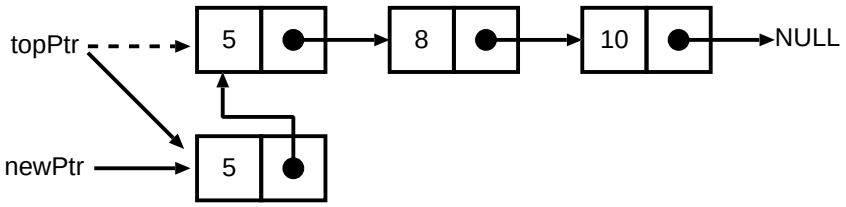


図 4: スタックに値をプッシュする

1. malloc() を呼び出して新しいノードを生成し，newPtr に確保したメモリ領域のアドレスを代入する (76 行目)
2. newPtr->data に挿入する値を代入し (79 行目)
3. newPtr->nextPtr に topPtr の値を代入し (80 行目)
4. topPtr を新たに作ったノードのアドレスを代入する (81 行目)。この操作により topPtr が新たなデータを指すようになる。

pop() はスタックから先頭ノードを削除する。main() では，pop() を呼び出す前に，スタックが空であるか否かを調べている (42 行目)。
pop() の手順は以下のとおり (図 5)。

1. topPtr を tmpPtr に代入する (92 行目)。これは後で free() するために用いる。
2. *pop_value に topPtr->data の値をセットし (93 行目),
3. topPtr->nextPtr の値を topPtr に代入 (94 行目),
4. tmpPtr の指すメモリ領域を解放 (95 行目),
5. topPtr の値を返す (96 行目)

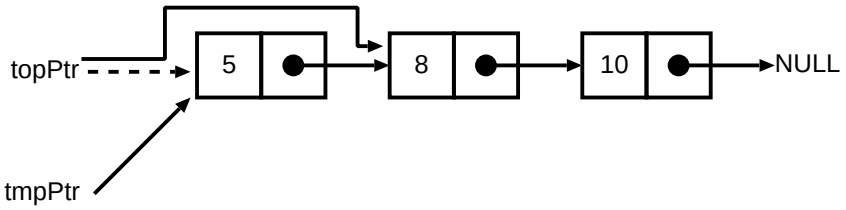


図 5: スタックから値をポップする

スタックには数多くの用途がある。たとえば，関数呼び出しが行われたとき，呼び出された関数は，呼び出し側の関数に戻る方法を知っていなければならない。そこで，戻りアドレスがスタックに積まれる。一連の関数呼び出しが起きた場合にいは，戻りアドレスが次々とスタックに積まれて行くので，Last-In, First-Out 方式によって各関数はそれぞれの呼び出した関数に戻ることができる。スタックを使えば，通常の非再帰呼び出しと同じ方式で再帰的な関数呼び出しを実現することができる。

## 7.4 キュー構造

もう一つのよく使われるデータ構造はキュー (待ち行列) 構造である。キューはスーパーなどのレジにできる行列に似ている。最初に並んだ人が最初にサービスを受け，後から来た人は行列の最後に並んでサービスを待つ。キューのノードは先頭からしか削除できず，最後にしかデータを挿入できない。このため FIFO(First-In, First-Out) 構造とも呼ばれる。キューにノードを追加する操作を enqueue，キューからノードを削除する操作を dequeue と呼ぶ。

キューはコンピュータシステムの中でいろいろな用途に使われる。通常 CPU が処理できるタスクは一時に一つだけであるので，ユーザの処理要求はキューにつながれる。ユーザがサービスを受けるにつれ，各ユーザの処理要求は一つずつ前に進む。キューの先頭にある処理要求が次にサービスを受けることになる。

キューはプリントのスプーリングにも使われる。東女のシステムのように複数のユーザが一台のプリンタを利用する環境では，プリントリクエストはキューに加えられる。プリントリクエストはそのリクエストがキューの先頭に来るまでキューの中で待たされる。

キューを実装したプログラム例を以下に示す。

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  typedef struct queue {
 5      int data;
 6      struct queue *nextPtr;
 7  } QUEUE;
 8
 9  QUEUE *enqueue(QUEUE *headPtr, int data);
10  QUEUE *dequeue(QUEUE *headPtr, int *data);
11  int isEmpty(QUEUE *queuePtr);
12  void print_queue(QUEUE *queuePtr);
13  void usage(void);
```

```
14
15 void usage(void)
16 {
17     printf("使い方 1: キューに値を追加する\n");
18     printf("       2: キューから値を削除する\n");
19     printf("       3: キューを印刷する\n");
20     printf("       4: 終了\n");
21 }
22
23 int main(void)
24 {
25     QUEUE *headPtr = NULL;
26     int data;
27     int menu;
28
29     usage();
30     printf("メニュー番号を入力してください:");
31     scanf("%d", &menu);
32
33     while(menu != 4) {
34         switch(menu) {
35         case 1:
36             printf("キュー追加する整数を入力してください:");
37             scanf("%d", &data);
38             headPtr = enqueue(headPtr, data);
39             print_queue(headPtr);
40             break;
41         case 2:
42             if ( !isEmpty(headPtr) ) {
43                 headPtr = dequeue(headPtr, &data);
44                 printf("キューから取り除いた値は %d です\n", data);
45             } else {
46                 printf("キューは空です\n");
47             }
48             break;
49         case 3:
50             if ( !isEmpty(headPtr) )
51                 print_queue(headPtr);
52             else
53                 printf("キューは空です\n");
54             break;
55         default:
56             printf("番号が不正です\n");
57             usage();
58             break;
```

```
59          }
60          printf("メニュー番号を入力してください:");
61          scanf("%d", &menu);
62      }
63
64      return 0;
65  }
66
67  int isEmpty(QUEUE *queuePtr)
68  {
69      return queuePtr == NULL;
70  }
71
72  QUEUE *enqueue(QUEUE *headPtr, int data)
73  {
74      QUEUE *newPtr, *tmpPtr;
75
76      newPtr = (QUEUE *)malloc(sizeof(QUEUE));
77
78      if ( newPtr != NULL ) {
79          newPtr->data = data;
80          newPtr->nextPtr = NULL;
81
82          if (isEmpty(headPtr)) {
83              headPtr = newPtr;
84          } else {
85              tmpPtr = headPtr;
86              while(tmpPtr->nextPtr != NULL) {
87                  tmpPtr = tmpPtr->nextPtr;
88              }
89              tmpPtr->nextPtr = newPtr;
90          }
91      } else {
92              printf("メモリが足りません\n");
93      }
94      return headPtr;
95  }
96
97  QUEUE *dequeue(QUEUE *headPtr, int *data)
98  {
99      QUEUE *tmpPtr;
100
101     if (headPtr == NULL)
102         return NULL;
103
```

```
104       *data = headPtr->data;
105       tmpPtr = headPtr;
106       headPtr = headPtr->nextPtr;
107       free(tmpPtr);
108
109       return headPtr;
110   }
111
112   void print_queue(QUEUE *queuePtr)
113   {
114       if (queuePtr == NULL) {
115           printf("キューは空です\n");
116       } else {
117           while ( queuePtr != NULL ) {
118               printf("%d-> ", queuePtr->data);
119               queuePtr = queuePtr->nextPtr;
120           }
121           printf("NULL\n\n");
122       }
123   }
```

enqueue() は引数としてキューの先頭ポインタのアドレスと挿入すべきデータを受け取り，キューの末尾にその値を挿入する。equeue の手続きは以下のとおり

1. malloc() を呼び出して新しいノードを生成し (76 行目)，newPtr に確保したメモリ領域のアドレスを代入する
2. キューが空の場合は newPtr を hedPtr に代入し (83 行目),
3. そうでない場合には，tmpPtr を headPtr で初期化し,
4. tmpPtr->nextPtr が NULL でない間 tmpPtr->nextPtr を tmpPtr に代入して一番最後のエントリを求める (86,87 行目)
5. キューの最後の要素に newPtr を付け加える

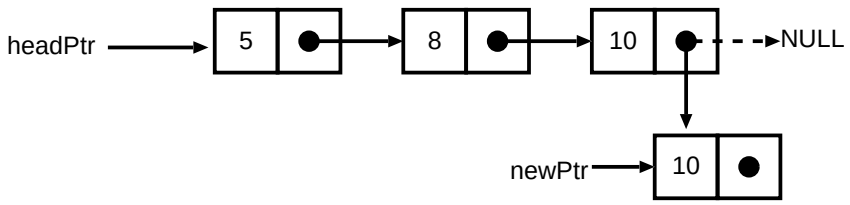


図 6: エンキューの操作

dequeue() は引数としてキューの先頭のポインタのアドレスを受け取って，int へのポインタにデータをセットする。dequeue の手続きは以下のとおり

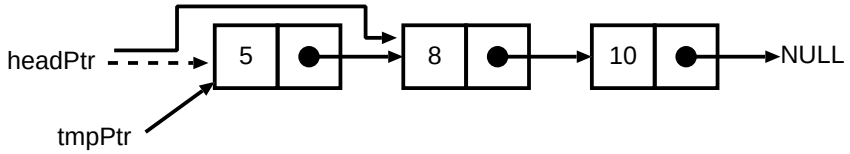


図 7: デキューの操作

1. headPtr が NULL であれば NULL を返す (101,102 行目)
2. headPtr->data の値を *data にセットする (104 行目)
3. tmpPtr に headPtr のアドレスを代入する (105 行目) これは，あとで free() を呼ぶためのものである。
4. headPtr->nextPtr のアドレスを headPtr に代入する (106 行目)

## 7.5 ツリー構造

連結リスト，スタック，キューは線形データ構造であるが，ツリー tree 構造は非線形な二次元データ構造である。二分木 binary tree 構造は通常 1 つのノードに 2 つのリンクポインタを持つ。ツリーの先頭ノードをルート (root 根) ノードと言い，ルートノードの各リンクは子ノードにつながっている。左子ノードは左サブツリーの先頭ノード，右子ノードは右サブツリーの先頭ノードを持つ。

ここでは，二分探索木と呼ばれる二分木を作る。重複したノード値をもたない二分探索木には,「どの左サブツリーのノード値も親ノードの値より小さく，どの右サブツリーのノード値も親ノードの値より大きい」という特徴がある。図 fig:btree は 6 個の値を持つ二分探索木である。一組のデータに対応する二分探索木の形状はデータを挿入する順序によって変化する。

サンプルプログラムを以下に示す。

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <time.h>
 4
 5  typedef struct tree {
 6      int data;
 7      struct tree *prevPtr;
 8      struct tree *nextPtr;
 9  } BTREE;
10
11  BTREE *insert(BTREE *Btree, int data);
12
13  void inOrder(BTREE *Btree);
14  void preOrder(BTREE *Btree);
15  void postOrder(BTREE *Btree);
16
17  void swap(int *a, int *b)
```

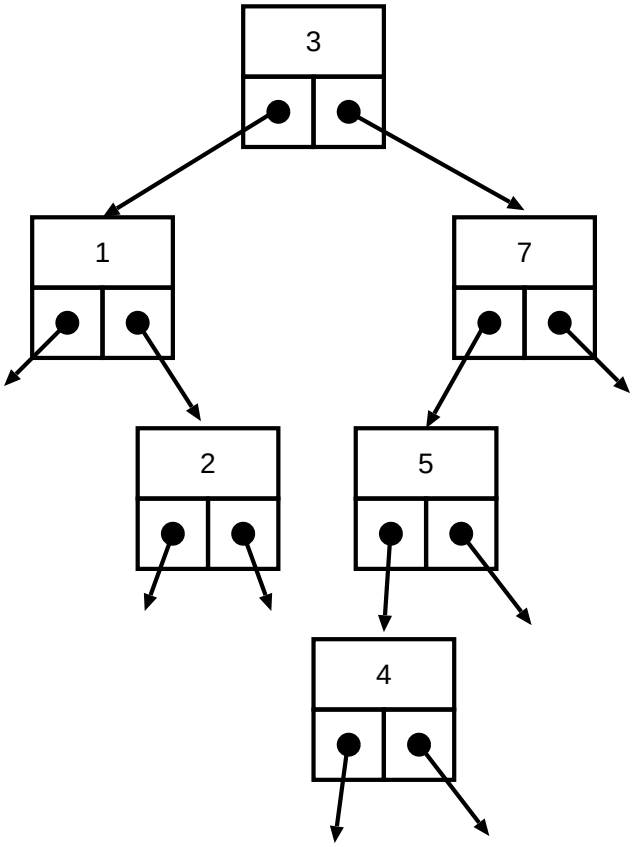


図 8: 二分木の構造

```
18  {
19      int tmp = *a;
20      *a = *b;
21      *b = tmp;
22  }
23
24  int main(void)
25  {
26      int i, item;
27      BTREE *tmpPtr, *rootPtr = NULL;
28      int data[15];
29      tmpPtr = rootPtr;
30
31      srand(time(NULL));
32
33      for (i=0; i<15; i++){
34          data[i]=i+1;
35      }
36      for (i=0; i<15; i++){
37          item = i + rand() % (15 - i);
```

```
38          swap(&data[i], &data[item]);
39      }
40
41      printf("2 分木に挿入する値:\n");
42      for (i=0; i<15; i++) {
43          printf("%3d", data[i]);
44          rootPtr = insert(rootPtr, data[i]);
45      }
46      printf("\n\n 先順走査\n");
47      preOrder(rootPtr);
48
49      printf("\n\n 中順走査\n");
50      inOrder(rootPtr);
51
52      printf("\n\n 後順走査\n");
53      postOrder(rootPtr);
54      printf("\n");
55
56      return 0;
57  }
58
59  BTREE *insert(BTREE *treePtr, int val)
60  {
61      if (treePtr == NULL) {
62          treePtr = (BTREE *)malloc(sizeof(BTREE));
63          if (treePtr != NULL) {
64              treePtr->data = val;
65              treePtr->prevPtr = NULL;
66              treePtr->nextPtr = NULL;
67          } else {
68              printf("メモリ不足で %d を挿入できません\n", val);
69          }
70      } else {
71          if ( val < treePtr->data ) {
72              treePtr->prevPtr = insert(treePtr->prevPtr, val);
73          } else if ( val > treePtr->data ) {
74              treePtr->nextPtr = insert(treePtr->nextPtr, val);
75          } else {
76              printf("重複\n");
77          }
78      }
79      return treePtr;
80  }
81
82  void preOrder(BTREE *treePtr)
```

```
 83  {
 84      if (treePtr != NULL) {
 85          printf("%3d", treePtr->data);
 86          inOrder(treePtr->prevPtr);
 87          inOrder(treePtr->nextPtr);
 88      }
 89  }
 90
 91  void postOrder(BTREE *treePtr)
 92  {
 93      if (treePtr != NULL) {
 94          inOrder(treePtr->prevPtr);
 95          inOrder(treePtr->nextPtr);
 96          printf("%3d", treePtr->data);
 97      }
 98  }
 99
100  void inOrder(BTREE *treePtr)
101  {
102      if (treePtr != NULL) {
103          inOrder(treePtr->prevPtr);
104          printf("%3d", treePtr->data);
105          inOrder(treePtr->nextPtr);
106      }
107  }
```

二分探索木にノードを追加する手順は以下のとおり

1. treePtr が NULL であれば malloc() を呼び出して新しいノードを生成し，treePtr に代入し (62 行目),
2. treePtr->data に値を代入し (64 行目)，treePtr->prevPtr と treePtr->nextPtr とに NULL を代入し (65,66 行目)，その treePtr を返す (79 行目)
3. treePtr が NULL でなく値 val が treePtr->data よりも小さければ，treePtr->prevPtr を引数として insert() を再帰的に呼び出し (72 行目)
4. val が treePtr->data よりも大きければ，treePtr->nextPtr を引数として insert() を再帰的に呼び出す (74 行目)

関数 inOrder() , preOrder(), postOrder() はそれぞれツリーを異なる順序でなぞりノード値を印刷する。

inOrder() 中順走査は,

1. inOrder() で左サブツリーをなぞる
2. そのノードの値を印刷する
3. inOrder() で右サブツリーをなぞる

図 8 を処理すると，
1, 2, 3, 4, 5, 7
となる。中順走査では，ノードが昇順に印刷される。二分探索木を生成する過程でデータがソートされるので，このような手法を二分木ソートという。

`preOrder()` 先順走査では以下の手順でノードをなぞる。

1. そのノードの値を印刷する
2. `preOrder()` で左サブツリーをなぞる
3. `preOrder()` で右サブツリーをなぞる

各ノードの値は，そのノードを最初に訪れた時点で処理される。あるノードの値を印刷した後，左サブツリーにある値が処理され，それが終わると右サブツリーにある値が処理される。図 8 先順走査で処理すると，
3, 1, 2, 7, 5, 4 となる。

`postOder()` 後順走査では以下の手順でツリーをなぞる。

1. `postOrder()` で左サブツリーをなぞる
2. `postOrder()` で右サブツリーをなぞる
3. そのノードの値を印刷する

各ノードの値は，両方のサブツリーを処理した後に処理される。図 8 先順走査で処理すると，
1, 2, 4, 5, 7, 3 となる。

キーとなる値と一致する値を二分木の中から高速に探索することができる。二分木が密集している場合，各レベルは一つ上のレベルよりも 2 倍のノード含んでいる。従って， n 個のノードからなる二分探索木は最大 $\log_2 n$ レベルあり，キー値と一致する値があるか否かを判断するには最大で $log_2 n$ 回比較すればよいことになる。たとえば 1000 ノードの二分探索木の場合 $2^{10} > 1000$ なのでたかだか 10 回比較すればよい。

**演習 7.2** 二分木を二次元的に印刷するプログラムを作れ。例えば，2 分木に挿入する値が，

```
 7 11  4  8  3 15  6  1 12  5 14 10  2  9 13
```

という順で与えられたとすると，

```
          15
                    14
                         13
               12
     11
               10
                    9
          8
 7
          6
               5
      4
          3
                    2
               1
```

と出力するプログラムを作成せよ。右端のリーフノードが画面の右端の列の最上位に現れ，ルートノードが画面の左端に現れる。各列はスペース 5 個ずつの間隔を開けて表示すること。`outputTree()` は引数として現在のノードへのポインタとノードの前に出力するスペースの数 (`totalSpaces`) を受け取る。ルートノードを画面の左端に表示するので `totalSpaces` は 0 から数える。この関数は中順走査の変形を使って二分木を表示する。アルゴリズムは以下のとおり，
現在のノードへのポインタが `NULL` でないあいだ，

1. 現在のノードの右サブツリーへのポインタと `totalSpaces + 5` を引数として，`outputTree()` を再帰的に呼び出す
2. `for` 文を使って 0 から `totalSpaces` までスペースを出力する
3. 現在のノードの値を出力する
4. 現在のノードへのポインタに，左サブツリーへのポインタをセットし，`totalSpaces` に 5 を足し込んで `outputTree()` を再帰的に呼び出す