ソフトウェア工学補助資料

2005年5セメスター

本資料について

本資料は、「ソフトウェア工学」における補助的な資料である 。主に、プログラムと課題を示すためとに用いる。

おおまかな書式は次のように 2 重枠で示す。

書式

プログラム内の記述は、次のように一重枠で示す。

```
int x;
```

ソースファイルは、次のよう左端に行番号付けながら罫線で挟んで示す。

リスト 1:HelloWorld.c

```
1  /*サンプルプログラム*/
2  #include <stdio.h>
3  int main()
4  {
5    printf("Hello world \n");
6    return 0;
7  }
```

実行方法と実行結果は、下のように丸枠で示す。なお、$は端末のコマンドプロンプトを表わす。

```
$./HelloWorld
HelloWorld
$
```

レポートについて

レポート課題は、Sxx-yy の形で提示する。一回のレポートには、複数の問題が含まれる。xx はレポートの回数を表わし、yy は同一レポートの問題番号を表わす。xx が同じ場合は、同じ締切である。

レポートの一部では、C 言語のソースコードの提出が求められる場合がある。その際には、3 セメスター時のプログラミング演習のときの提出の仕組を利用することができる。例えば、Sxx.yy の問題において、hogehoge.c を提出したい場合には、次のように行なえばよい。

```
$submit  Sxx  yy hogehoge.c
```

なお、ソースコード以外は紙として提出すること。プログラミング演習室以外でプログラミングを行なう人は、ソースコードをプリントアウトして提出して下さい。

プログラムの実行時間計測

ここでは、Unix 上で、C 言語のプログラムを動作させたときの時間計測の方法を 2 つ示す。

0.1 time コマンドによる方法

Unix には、プログラムの実行時間計測のために time コマンドがある。この time コマンドでプログラムの実行時間を計測するは次のように行なう。

```
time 実行コマンド
```

で実行時間を計測できる。

```
$time  ./HelloWorld
Hello world

real    0m0.002s
user    0m0.000s
sys     0m0.000s
$
```

ここで、real は実時間を表わし、user はプログラム中の自分で用いた部分 (ユーザ部分) を表わし、sys はプログラム中のシステムが用いた部分 (システム部分) を表わす[1]。

リスト 1 に、時間がある程度かかるプログラムを示す。

リスト 1:time_test.c

```
1   /*時間計測のサンプルプログラム*/
2   #include <stdio.h>
3   #define N 10000  /*繰り返し回数の制御*/
4
5   int main()
6   {
7     int i=0;
8     int j=0;
9
```

[1]なお、通常は、他のプログラムも動作しているので、user + sys≠real である。

```
10      for(i=0;i<N;i++)
11        {
12          for(j=0;j<N;j++)
13            {
14              printf("");
15              /*printf("a");*//*システム時間の増加*/
16            }
17        }
18      printf("繰り返し終了\n");
19      return 0;
20    }
```

このリスト 1 の計測を time コマンドで行なう。

```
$ time ./time_test
繰り返し終了

real    0m5.710s
user    0m5.680s
sys     0m0.010s
$
```

課題

0.1.1 リスト 1 中のコメントをはずし、システムの時間を計測せよ。

0.2 ソースコードに埋め込む方法

time コマンドでは、プログラム全体の実行時間しか計れない。しかし、プログラムの一部だけに費やされた時間を計測したいこともよくある。そこで、ソースコード内に次のように、clock() ライブラリ関数を埋め込むことで実行時間を計測できる。

```
clock_t start
clock_t end
～
start=clock();
計測部分
end=clock();
測定時間=(end-start)/CLOCKS_PER_SEC
```

なお、clock() はプログラム開始時からの経過時間を求める関数である。CLOCKS_PER_SEC

は clock() 関数の戻り値の 1 秒当たりの数を示すマクロである。clock() の戻り値を CLOCKS_PER_SECで割ることによって秒単位の値が求まる。また、clock_t型には、(double) や (int) のキャスト演算子を用いることができる。これらを利用するには、time.h ヘッダをインクルードする必要がある。したがって、上の 2 重枠内のような記述で、clock() で囲まれた部分に費やされたプロセッサ時間を求めることができる。
リスト 2:count_time.c

```
1  /*時間計測のサンプルプログラム*/
2  #include <stdio.h>
3  #include <time.h>
4
5  #define N 10000  /*繰り返し回数の制御*/
6
7  void non_count();/*非計測部分*/
8  void count();/*計測部分*/
9
10 int main()
11 {
12   clock_t start;
13   clock_t end;
14   double sec;
15
16   non_count();
17   printf("非計測部分終了\n");
18
19   start=clock();
20   count();
21   end=clock();
22
23   sec=(double)(end-start)/CLOCKS_PER_SEC;
24
25   printf("計測部分終了\n");
26   printf("計測部分は、%f 秒です。\n",sec);
27
28   return 0;
29 }
30
31 /*非計測部分*/
```

```
32  void non_count()
33  {
34    int i=0;
35    int j=0;
36    for(i=0;i<N;i++)
37      {
38        for(j=0;j<N;j++)
39          {
40            printf("");
41          }
42      }
43    return;
44  }
45
46  /*計測部分*/
47  void count()
48  {
49   int i=0;
50   int j=0;
51    for(i=0;i<N;i++)
52      {
53        for(j=0;j<N;j++)
54          {
55            printf("");
56          }
57      }
58    return;
59  }
```

```
$time ./count_time
非計測部分終了
計測部分終了
計測部分は、4.690000 秒です。

real    0m11.084s
user    0m9.690s
sys     0m0.010s
$
```

課題

0.2.1 リスト 2 において、計測範囲や計測中関数内部等を色々変化させて、`time` コマンドの計測時間と比較せよ。

0.2.2 リスト 2 はプログラミング演習のスタイルに違反している部分がある。どの部分かを発見せよ。

レポート 課題 S0

提出締切：2005/04/28/(木曜日)
提出先：GI511 レポート提出箱

S0-1 リスト 1 中の`#define` によって、`N` の割当てを変更し、`N` を横軸、時間を縦軸としてグラフにまとめよ。なお、この課題の時間計測は `time` コマンドで行なうこと。

S0-2 上のレポート課題 S0-1 において、時間の変化が `N` の 2 乗に比例するかどうかを考察せよ。

S0-3 同一のソースファイル内に、$O(n^3)$ 時間の関数と $O(n^2)$ 時間の関数を作成せよ。ソースコードをとして提出する。提出は下記のように行なう。

```
submit S0 3 ファイル名
```

S0-4 上のレポート問題 S0-3 において、2 つの関数それぞれの経過時間を `clock()` 関数を用いて計測せよ。

第1章　アルゴリズム入門

1.1　最大値を求めるアルゴリズム (線形時間アルゴリズム)

リスト 3 に、最大値を求めるプログラムを示す。

リスト 3:find_max.c

```
1   /*find_max.c 最大値を求めるサンプルプログラム*/
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   #define N 100       /*データ数*/
6   #define W 1000000   /*1 ステップを遅くするため重み*/
7
8   void make_data();/*ランダムな整数データの生成*/
9   void print_data();/*データの表示*/
10  void weight();/*1 ステップを遅くする関数*/
11
12  int Data[N];/*データ*/
13
14  int main()
15  {
16    int max=0;
17    int i;
18
19    make_data();
20
21    max=Data[0];
22    for(i=0;i<N;i++)
23      {
24        weight();
25
26        if(Data[i]>max)
```

```
27          {
28            max=Data[i];
29          }
30      }
31
32    print_data();
33
34    printf("最大値= %10d \n",max);
35
36    return 0;
37  }
38
39  /*
40  ランダムな整数データの生成
41  グローバル変数 Data に設定
42  */
43  void make_data()
44  {
45    int i;
46    srand(time(NULL));/*乱数の生成*/
47
48    for(i=0;i<N;i++)
49      {
50        Data[i]=rand();
51      }
52    return;
53  }
54
55  /*データの表示。Data を標準出力へ出力*/
56  void print_data()
57  {
58    int i;
59    for(i=0;i<N;i++)
60      {
61        printf("%10d ",Data[i]);
62        if(i%5==4)
63          {
64            printf("\n");
65          }
```

```
66        }
67
68      return;
69    }
70
71    /*1 ステップを遅くする関数。マクロ W によって制御*/
72    void weight()
73    {
74      int i;
75      for(i=0;i<W;i++)
76        {
77          printf("");
78        }
79      return;
80    }
```

このリスト 3 において、weight() は 1 ステップを遅くする関数であり、アルゴリズムの動作には本質に関わらない。この関数 weight() は、入力サイズとステップ数の関係を実時間を計測して調べるためだけに利用される。

課題

1.1.1 リスト 3 が、$O(n)$ の時間計算量であることを確認せよ。その際、W の割当てを変更しても、$O(n)$ 時間すなわち線形時間であることを確認すること。

1.2 最大公約数 (多項式時間アルゴリズム 対 対数時間アルゴリズム)

素朴なアルゴリズムを用いて、最大公約数を求めるプログラムを、リスト 4 に示す。

リスト 4:naive_gcd.c

```
1   /*素朴な方法で最大公約数を求めるプログラム*/
2   #include <stdio.h>
3   #define A 123456789  /*数 A*/
4   #define B 345678912  /*数 B*/
5
```

```
6   int min(int a,int b);/*最小値を求める関数*/
7   int find_gcd(int a,int b);/*最大公約数を求める関数*/
8
9   int main()
10  {
11    int gcd=0;  /*最大公約数*/
12
13    printf("%d と %d の最大公約数を求めます。\n",A,B);
14    gcd=find_gcd(A,B);
15    printf("最大公約数は、%d です。\n",gcd);
16
17    return 0;
18  }
19
20  /*
21    a と b の最大公約数を求める関数
22    引数:a,b
23  */
24  int find_gcd(int a,int b)
25  {
26    int i=0;   /*ループカウンタ*/
27    int gcd=0; /*最大公約数*/
28    int small=0;/*a と b の小さい方*/
29
30    small=min(a,b);
31
32    for(i=small;i>0;i--)
33      {
34        if((a%i==0)&&(b%i==0))
35          {
36            /*両方で割り切れるので公約数*/
37            gcd=i;
38            break;
39          }
40      }
41
42    return gcd;
43  }
44
```

```
45  /*a と b との小さい方を返す関数*/
46  int min(int a,int b)
47  {
48    if(a<=b)
49      {
50        return a;
51      }
52    else
53      {
54        return b;
55      }
56  }
```

ユークリッドの互除法を用いて、最大公約数を求めるプログラムを、リスト 5 に示す。

リスト 5:euclid_gcd.c

```
1   /*ユークリッドの互除法で最大公約数を求めるプログラム*/
2   #include <stdio.h>
3   #define A 123456789  /*数 A*/
4   #define B 345678912  /*数 B*/
5
6   int mix(int a,int b);/*最小値を求める関数*/
7   int max(int a,int b);/*最大値を求める関数*/
8   int euclid(int a,int b);/*最大公約数を求める関数*/
9
10  int main()
11  {
12    int gcd=0;  /*最大公約数*/
13
14    printf("%d と %d の最大公約数を求めます。\n",A,B);
15    gcd=euclid(A,B);
16    printf("最大公約数は、%d です。\n",gcd);
17
18    return 0;
19  }
20
21  /*
```

```
22    ユークリッドの互除法で,a と b の最大公約数を求める関数
23    引数:a,b
24  */
25  int euclid(int a,int b)
26  {
27    int big=0;   /*大きい方*/
28    int small=0; /*小さい方*/
29    int remainder=0; /*余り*/
30
31    big=max(a,b);
32    small=min(a,b);
33
34    remainder=big%small;
35    while(remainder!=0)
36      {
37        big=small;
38        small=remainder;
39        remainder=big%small;
40      }
41
42    return small;
43  }
44
45  /*a,b の小さい方を返す関数*/
46  int min(int a,int b)
47  {
48    if(a<=b)
49      {
50        return a;
51      }
52    else
53      {
54        return b;
55      }
56  }
57
58  /*a,b の大きい方を返す関数*/
59  int max(int a,int b)
60  {
```

```
61      if(a>b)
62        {
63          return a;
64        }
65      else
66        {
67          return b;
68        }
69    }
```

課題

1.2.1 リスト 4 中の、A,B を色々変化させてプログラムを実行せよ。

1.2.1 リスト 5 中の、A,B を色々変化させてプログラムを実行せよ。

1.3 フィボナッチ数列 (指数時間アルゴリズム 対 多項式時間アルゴリズム)

リスト 6 に、再帰を用いてフィボナッチ数列を求める関数と、配列を用いてフィボナッチ数列を求める関数を示す。

リスト 6:swich_fibo.c

```
1   /*
2   フィボナッチ数列を求める関数を用いて、
3   指数時間と多項式時間を体験するサンプルプログラム
4   */
5   #include <stdio.h>
6   #define N 40  /*第 N 項*/
7
8   double fibo_rec(double n);/*再帰的な関数*/
9   double fibo_array(double n);/*配列を用いた関数*/
10
11  int main()
12  {
```

```
13    double fibo_num=0.0; /*フィボナッチ数の結果*/
14
15    /*下のどちらかを実行する。*/
16    fibo_num=fibo_rec((double)N);
17    /* fibo_num=fibo_array((double)N);*/
18
19    return 0;
20  }
21
22  /*フィボナッチ数列を求める再帰的な関数*/
23  double fibo_rec(double n)
24  {
25    if(n<1.0)
26      {
27        return 0.0;
28      }
29    else if(n<2.0)
30      {
31        return 1.0;
32      }
33    else
34      {
35        return (fibo_rec(n-1.0)+fibo_rec(n-2.0));
36      }
37  }
38
39  /*
40    これまでの項を保存して、
41    フィボナッチ数列を求める関数
42  */
43  double fibo_array(double n)
44  {
45    double f[N];/*数列を蓄える配列*/
46    int i;
47
48    f[0]=0.0;
49    f[1]=1.0;
50    for(i=2;i<N;i++){
51      f[i]=f[i-1]+f[i-2];
```

```
52      }
53      return f[N-1]+f[N-2];
54    }
```

課題

1.3.1 リスト 6 中の`#define`によって、`N`の割当てを変更し、2 つの関数の実行時間を計測せよ。

レポート 課題 S1

提出締切:2005/05/20(金)
提出先：講義前に提出するか、GI511 レポート提出箱に提出する。

S1-1 リスト 3 中の`#define`によって、`N`の割当てを変更し、`N`を横軸、時間を縦軸としてグラフにまとめよ。

S1-2 リスト 4 とリスト 5 に対して、同じように`A,B`を変化させた場合、その実行時間を比較考察せよ。なお、プログラム中で、`A,B`を変化させるようにしてもかまわない。(その時には、プログラムのソースコードも提出すること。)

S1-3 ユークリッドの互除法は、再帰的な関数を用いても実現できる。ユークリッドの互除法を再帰的に実現した C 言語の関数を作成せよ。(ソースを提出すること。)

S1-4 リスト 6 中の関数`fibo_rec(N)`および`fibo_array(N)`の時間量をそれぞれ $O-$ 記法で示せ。(簡単な導出も示すこと。)

第2章　多項式の計算

2.1　べき乗の計算

リスト 7 に、べき乗 (x^n) を求めるプログラムを示す。

リスト 7:mypow.c

```
1   /*ベキ乗を計算するサンプルプログラム*/
2   #include <stdio.h>
3
4   #define EPS (1.0e-5) /*微小数*/
5   #define N 16384 /*ベキ (2 のベキに指定する)*/
6   #define W 10000 /*1 ステップの重み*/
7
8   void weight();/*1 ステップを遅くする関数*/
9   double naive_pow(double x,int n);/*素朴な方法でべき乗を求める関数*/
10  double fast_pow(double x,int n);/*ベキ乗を高速に求める関数*/
11
12  int main()
13  {
14    double x=1.0+EPS;  /*x*/
15    double x_n=0.0;/*x の N 乗*/
16
17    x_n=naive_pow(x,N);
18    /*x_n=fast_pow(x,N);*/
19
20    printf("x   =%20.10f の\n",x);
21    printf("%10d 乗は\n",N);
22    printf("%20.10f\n",x_n);
23
24    return 0;
25  }
26
```

```
27  /*x の n 乗を求める関数*/
28  double naive_pow(double x,int n)
29  {
30    int i=0;
31    double x_n=1.0;/*x の n 乗*/
32
33    x_n=x;
34    for(i=1;i<n;i++)
35      {
36        weight();
37        (x_n)=(x_n)*x;
38      }
39
40    return x_n;
41  }
42
43  /*ベキ乗を高速に求める関数, ただし、N は 2 の k 乗とする。*/
44  double fast_pow(double x,int n)
45  {
46    int i=1;
47    double x_i;/*x の i 乗を蓄える変数*/
48
49    x_i=x;
50    for(i=1;i<N;i=2*i)
51      {
52        weight();
53        x_i=x_i*x_i;
54      }
55
56    return x_i;
57  }
58
59  /*1 ステップを遅くする関数。マクロ W によって制御*/
60  void weight()
61  {
62    int i;
63    for(i=0;i<W;i++)
64      {
65        printf("");
```

```
66        }
67      return;
68    }
```

課題

2.1.1 リスト 7 において、パラメータ (N,W,EPS) の値を色々と変化させて、関数 naive_pow() および関数 fast_pow() を動作させよ。

2.2 ホーナーの方法

リスト 8 に、素朴な方法で、多項式

$$f(x) = a_0 + a_1 * x + \cdots + a_n * x^n = \sum_{i=0}^{n} a_i x^i$$

を計算するプログラムを示す。

リスト 8:naive_poly.c

```
1   /*素朴な方法で、多項式を計算するサンプルプログラム*/
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   #define MAX 20 /*係数の絶対値の最大値*/
6   #define N 9999 /*最高次数*/
7
8   int A[N+1];/*係数 a_0 ～ a_n*/
9
10  void  make_keisu();/*係数の生成 */
11  void  print_keisu();/*係数の表示*/
12  double  make_input();/*x の生成*/
13
14  double naive_pow(double x,int n);/*x の n 乗を求める関数*/
15  double f(double x);/*f(x) の計算*/
16
17  int main()
```

```
18  {
19    double x=0.0;  /*x*/
20    double fx=0.0;/*f(x)*/
21
22    make_keisu();
23    /*print_keisu();*/
24    x=make_input();
25    printf("x   =%20.10f\n",x);
26
27    fx=f(x);
28
29    printf("f(x)=%20.10f\n",fx);
30
31    return 0;
32  }
33
34  /*
35    係数の生成
36    -MAX～MAX までの整数
37  */
38  void  make_keisu()
39  {
40    int i=0;
41    srand(time(NULL));/*乱数系列の初期化 (乱数の種の設定)*/
42
43    for(i=0;i<=N;i++)
44      {
45        /*乱数の生成*/
46        A[i]=(int)((2.0*MAX)*rand()/(RAND_MAX + 1.0)-MAX);
47      }
48    return;
49  }
50
51  /*f(x) の係数の表示*/
52  void  print_keisu()
53  {
54    int i=0;
55    for(i=0;i<=N;i++)
56      {
```

```
57          printf("%4d",A[i]);
58          if(i%10 == 9)
59            {
60              printf("\n");
61            }
62        }
63      return;
64    }
65
66    /*0～1 の実数生成。*/
67    double make_input()
68    {
69      double x;
70      srand(time(NULL));
71
72      x=(double) rand()/(RAND_MAX + 1.0);
73      /*1.0 を加えることで 0 で割ることが無いようにしている。*/
74
75      return x;
76    }
77
78    /*x の n 乗を求める関数*/
79    double naive_pow(double x,int n)
80    {
81      int i=0;
82      double x_n=1.0;/*x の n 乗*/
83
84      for(i=0;i<n;i++)
85        {
86          (x_n)=(x_n)*x;
87        }
88
89      return x_n;
90    }
91
92    /*f(x) の計算*/
93    double f(double x)
94    {
95      int i=0;
```

```
96      double fx=0.0;/*f(x)*/
97
98      for(i=0;i<=N;i++)
99        {
100         (fx)=(fx)+(A[i]*naive_pow(x,i));
101       }
102
103     return fx;
104   }
105
106
107
```

課題

2.2.1 リスト 8 に対して、パラメータ (N) の値を色々と変化させて、プログラムを実行させよ。

ホーナーの方法を用いて、多項式を求めるプログラムを、リスト 9 に示す。なお、ホーナーの方法は、

$$f(x) = a_0 + a_1 * x + \cdots + a_n * x^n = (a_0 + (a_1 + (\cdots (a_{n-2} + (a_{n-1} + (a_n) * x) * x) \cdots) * x) * x) * x)$$

という計算式から多項式の値を求める方法である。

リスト 9:horner_poly.c

```
1   /*ホーナーの方法で多項式を計算するサンプルプログラム*/
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   #define MAX 20 /*係数の絶対値の最大値*/
6   #define N 9999 /*最高次数*/
7
8   int A[N+1];/*係数 a_0 ～ a_n*/
9
10  void  make_keisu();/*係数の生成 */
11  void  print_keisu();/*係数の表示*/
```

```
12  double  make_input();/*x の生成*/
13
14  double horner(int k,double x);/*f_k(x) の計算*/
15  double f(double x);/*f(x) の計算*/
16
17  int main()
18  {
19    double x=0.0;  /*x*/
20    double fx=0.0;/*f(x)*/
21
22    make_keisu();
23    /*print_keisu();*/
24    x=make_input();
25    printf("x   =%20.10f\n",x);
26
27    fx=f(x);
28
29    printf("f(x)=%20.10f\n",fx);
30
31    return 0;
32  }
33
34  /*
35    係数の生成
36    -MAX～MAX までの整数
37  */
38  void  make_keisu()
39  {
40    int i=0;
41    srand(time(NULL));/*乱数系列の初期化 (乱数の種の設定)*/
42
43    for(i=0;i<=N;i++)
44      {
45        /*乱数の生成*/
46        A[i]=(int)((2.0*MAX)*rand()/(RAND_MAX + 1.0)-MAX);
47      }
48    return;
49  }
50
```

```
51  /*f(x) の係数の表示*/
52  void  print_keisu()
53  {
54    int i=0;
55    for(i=0;i<=N;i++)
56      {
57        printf("%4d",A[i]);
58        if(i%10 == 9)
59          {
60            printf("\n");
61          }
62      }
63    return;
64  }
65
66  /*0〜1 の実数生成。*/
67  double make_input()
68  {
69    double x;
70    srand(time(NULL));
71
72    x=(double) rand()/(RAND_MAX + 1.0);
73
74    return x;
75  }
76
77  /*f_k(x) を求める関数*/
78  double horner(int k,double x)
79  {
80    if(k==0)
81      {
82        return ((double)A[N]);
83      }
84    else
85      {
86        return ((double)A[N-k]+(horner(k-1,x))*x);
87      }
88  }
89
```

```
90   /*f(x) の計算*/
91   double f(double x)
92   {
93     return horner(N,x);
94   }
```

課題

2.2.1 リスト 9 に対して 、パラメータ (N) の値を色々と変化させて、プログラムを実行させよ。

レポート 課題 S2

提出締切：2005/06/10(Fri.)
提出先：講義直前に提出するか、GI511 レポート提出箱に提出する。

S2-1 リスト 7 において、関数 naive_pow() および関数 fast_pow() の入力サイズと時間の関係をグラフとしてまとめよ。

S2-2 リスト 7 中の fast_pow() は、2 のベキ乗に対してしか正しくベキ乗を求めることができない。そこで、すべての自然数に対して、ベキ乗を高速に求めるアルゴリズムを作成せよ。ただし、このアルゴリズムは、$O((\log n)^2)$ の最悪時間計算量で x^n を求められるようにすること。アルゴリズムは、自然言語、擬似コード、フローチャート等のいずれを用いて表現してもかまわない。(この課題は、プログラムを作成する必要は無い[1]。)

S2-3 リスト 8 およびリスト 9 において、入力サイズと時間の関係をグラフとしてまとめよ。

S2-4 ホーナーの方法は再帰ではなくて、繰り返しを用いても記述することができる。ホーナーの方法を繰り返しを用いて実現したプログラムを作成せよ。

[1] プログラムを作成しても良い。その際は、C 言語のビット演算処理を調べると良い

第3章　ソーティング

ソースの分割

ソートのプログラムを示す前に、分割コンパイルの方法について述べる。

ある程度以上の規模のプログラムを作成するには、ソースコードをいくつかのファイルに分割して作成することが必要になる。ここでは、Unix系のOSで用いられるソースコードの分割の方法と、分割されたソースコードから一つの実行ファイルを作成するための方法を示す。

分割の一つの手段としては、#includeによるソースの取り込みの方法がある。プログラム中で共通に用いられる、マクロ、関数のプロトタイプ宣言、グローバル変数、等をヘッダファイルとしてまとめておけば、そのヘッダファイルを分割された個々のソースファイルに#includeで読み込むことで、見透し良くプログラミングできる。

ここでは、ソートで用いられる各種宣言類をまとめたヘッダファイルを、リスト10に示す。

リスト 10:sort.h

```
1   /*ソート関連のヘッダ*/
2   /*マクロ*/
3   #define TRUE 1   /*真*/
4   #define FALSE 0   /*偽*/
5
6   #define  N  10  /*データ数*/
7   #define  W  100 /*重み*/
8
9
10  /*関数のプロトタイプ宣言*/
11  void weight();/*1ステップを遅くする*/
12  void make_data();/* 配列Dataの値を設定する。*/
13  void print_data();/* 配列Dataの値を表示する。*/
14
15  void swap(double *a,double *b); /*値を交換する。swap(&a,&b)で呼びだす。*/
16
```

```
17  /*低速ソート*/
18  void bubble_sort();/*バブルソート*/
19  void selection_sort();/*選択ソート*/
20  void insertion_sort();/*挿入ソート*/
21  /*高速ソート (平均時間)*/
22  void quick_sort();/*クイックソート*/
23  /*高速ソート (最悪時間) */
24  void merge_sort();/*マージソート*/
25  void heap_sort();/*ヒープソート*/
26  void heap_sort2();/*ヒープソート*/
27
28
29  /*データを蓄えるグローバル変数*/
30  double Data[N];
```

このヘッダファイルは、次のようにして各ソースファイルに取り込むことができる。

#include “ヘッダファイル名”

```
#include "sort.h"
```

なお、#include を用いれば、「.c」の拡張子を持つ他のソースコードも取り込むことができる。

次に、リスト 11 に、ソート本体とは分離した関数類を示す。このプログラムには、main() 関数が無いので、そのままでは実行ファイルを作成できない。しかし、次のように -c オプションを用いて、コンパイルすることによって、**オブジェクトコード** を作成できる。なお、複数のオブジェクトコードが **リンク** されることによって、一つの実行ファイルが生成される。

gcc -c ソースファイル名 -o オブジェクトファイル名

```
$ls
sort.h sort_util.c
$gcc -c sort_util.c -o sort_util.o
$ls
sort.h sort_util.c sort_util.o
$
```

このコンパイルによって、sort_util.o というオブジェクトファイルが生成されいるのがわかる。この sort_util.o はそのままでは実行できないのだが、main 関数を持つソースコードから生成されたオブジェクトコードにリンクされることによって、sort_util.c で定義された関数が実行される。

リスト 11:sort_util.c

```
1  /*ソート関連の関数定義*/
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include"sort.h"
5
6  /*1 ステップを遅くする関数*/
7  void weight()
8  {
9    int i;
10   for(i=0;i<W;i++)
11     {
12       printf("");
13     }
14 }
15
16 /*
17   配列 Data の値を設定する関数
18  各値は、(0,1) の実数
19 */
20 void make_data()
21 {
22   int i;
23   srand(time(NULL));/*乱数系列の初期化*/
24
25   for(i=0;i<N;i++)
26     {
27       Data[i]=(double)rand()/(RAND_MAX+1.0);
28     }
29   return;
30 }
31
32 /* 配列 Data の中身を表示する関数*/
33 void print_data()
34 {
35   int i;
36   for(i=0;i<N;i++)
37     {
38       printf("%12.8f",Data[i]);
39       if(i%5 ==4)
```

```
40          {
41            printf("\n");
42          }
43       }
44     printf("\n");
45     return;
46   }
47
48   /*変数 a と変数 b の中身を交換する関数。
49     swap(&a,&b) として呼びだす。*/
50   void swap(double *a,double *b)
51   {
52     double tmp;
53     tmp=*a;
54     *a=*b;
55     *b=tmp;
56
57     return;
58   }
```

main 関数を持つソースコードをリスト 12 に示す。ソートの各プログラムは、すべてこのプログラムにリンクして利用することにする。

リスト 12:test_sort.c

```
1   /*ソート関数をテストするプログラム*/
2   /*#include <time.h>*/
3   #include <stdio.h>
4   #include "sort.h"
5
6   int main()
7   {
8     make_data();
9     print_data();
10    /*bubble_sort();*/
11    /*  selection_sort();*/
12    /*insertion_sort();*/
13    /*quick_sort();*/
```

```
14      /* merge_sort();*/
15      /* heap_sort();*/
16       heap_sort2();
17
18      printf("\n\n\n\n\n");
19      print_data();
20
21      return 0;
22    }
```

まず、test_sort.cからオブジェクトファイルを作成する。

```
$ls
sort.h sort_util.c sort_util.o test_sort.c
$gcc -c test_sort.c -o test_sort.o
$ls
sort.h sort_util.c sort_util.o test_sort.c test_sort.o
$
```

これらのオブジェクトファイルから実行ファイルを作成するには、次のように、オブジェクトファイルを指定して、gccコマンドを実行する。

gcc 複数のオブジェクトファイル名 -o 実行ファイル名

```
$gcc sort_util.o test_sort.o -o test_sort
```

この一連の手順により、実行ファイル (test_sort) が生成される。これらの一連の手順は、makeによっても実現できる。その際のMakefileの中身は、次のリスト 24 のように記述する。

リスト 13:Makefile

```
1   CC=gcc
2   objects = test_sort.o sort_util.o
3
4   all:test_sort
5   test_sort:$(objects)
6   $(objects):sort.h
```

分割されたソースファイルが増えた場合には、必要なぶんだけオブジェクトコード名を並らべれば良い。本章で最後に得られるMakefileは次のようになるはずである。リスト

14:Makefile

```
1  CC=gcc
2  objects = test_sort.o sort_util.o bubble_sort.o selection_sort.o \
3          insertion_sort.o quick_sort.o merge_sort.o heap_sort.o heap_sort2.o
4
5  all:test_sort
6  test_sort:$(objects)
7  $(objects):sort.h
```

3.1　バブルソート

バブルソートの実装を、リスト 15 に示す。

リスト 15:bubble_sort.c

```
1  /*バブルソート*/
2  #include <stdio.h>
3  #include "sort.h"
4
5  void bubble_sort()
6  {
7    int i;  /*パスの回数を制御*/
8    int j;  /*パスの毎の繰り返し回数を制御*/
9
10   for(i=0;i<N;i++)
11     {
12       for(j=N-1;j>i;j--)
13         {
14           weight();
15           if(Data[j]<Data[j-1])
16             {
17               swap(&Data[j],&Data[j-1]);
18             }
19         }
20     }
21
22   return;
23 }
```

3.2 選択ソート

選択ソートの実装を、リスト 16 に示す。

リスト 16:selection_sort.c

```
1  /*選択ソート*/
2  #include <stdio.h>
3  #include "sort.h"
4
5  /*プロトタイプ*/
6  int find_min(int left,int right);/*最小値の添字を見つける。*/
7
8  /*選択ソート本体*/
9  void selection_sort()
10 {
11   int i=0;  /*パスの回数を制御*/
12   int min_index=0;/*最小値を保持している Data の添字*/
13
14   for(i=0;i<N;i++)
15     {
16       min_index=find_min(i,N-1);
17       swap(&Data[i],&Data[min_index]);
18     }
19
20   return;
21 }
22
23 /*
24 Data[left] から Data[right] の間で、
25 最小値の添字を見つける。
26 */
27 int find_min(int left,int right)
28 {
29   int min_index=left;
30   int j=left;
```

```
31
32    min_index=left;
33    for(j=left+1;j<=right;j++)
34      {
35        weight();
36        if(Data[min_index]>Data[j])
37          {
38            min_index=j;
39          }
40      }
41
42    return min_index;
43  }
```

3.3　挿入ソート

挿入ソートの実装を、リスト 17 に示す。

リスト 17:insertion_sort.c

```
1   /*挿入ソート*/
2   #include <stdio.h>
3   #include "sort.h"
4
5   /*プロトタイプ*/
6   int find_pos(int left,int right);/*Data[right] の挿入位置を求める。*/
7   void insert(int pos,int right);/*Data[right] を Data[pos] に挿入する。*/
8
9   /*選択ソート本体*/
10  void insertion_sort()
11  {
12    int i=0;  /*パスの回数を制御*/
13    double in;
14    int pos=0;
15
16    for(i=1;i<N;i++)
17      {
```

```
18          pos=find_pos(0,i);
19          insert(pos,i);
20        }
21
22      return;
23    }
24
25    /*Data[right] の挿入位置を求める。*/
26    int find_pos(int left,int right)
27    {
28      int j=left;
29
30      for(j=left;j<=right;j++)
31        {
32          weight();
33          if(Data[j]>Data[right])
34            {
35              break;
36            }
37        }
38
39      return j;
40    }
41
42    /*Data[right] を Data[pos] に挿入する。*/
43    void insert(int pos,int right)
44    {
45      int k=right-1;
46
47      for(k=right-1;k>=pos;k--)
48        {
49          weight();
50          swap(&Data[k],&Data[k+1]);
51        }
52
53      return;
54    }
```

3.4　クイックソート

クイックソートの実装を、リスト 18 に示す。なお、クイックソートに於て、ピボットには右端の要素 (分割区間の最大添字の要素) を用いるようにしてある。

リスト 18:quick_sort.c

```
1  /*クイックソート関数*/
2  #include <stdio.h>
3  #include "sort.h"
4
5  /*プロトタイプ宣言*/
6  void q_sort(int left, int right);/*クイックソート本体*/
7  int partition(int left,int right);/*分割*/
8
9  /***********************関数定義***********************************/
10 /*他のソート形式に合わせるための関数*/
11 void quick_sort(){
12   q_sort(0,N-1);
13   return;
14 }
15
16 /* クイックソート本体。再帰を用いる。*/
17 void q_sort(int left, int right)
18 {
19   int pos;/*分割位置*/
20
21   if(left>=right)
22     {
23       return;
24     }
25   else
26     {
27       pos=partition(left,right);
28       q_sort(left,pos-1);
29       q_sort(pos+1,right);
30
31       return;
32     }
```

```
33  }
34
35  /*
36  Data[right] をピボットとして、
37  分割する関数。
38  */
39  int partition(int left,int right)
40  {
41    double pivot=Data[right];
42    int inc;  /*Data の添字。増加させながら調べる変数。*/
43    int dec;  /*減少させながら調べる変数。*/
44
45    /*基礎*/
46    if(left>=right)
47      {
48        return left;
49      }
50
51    /*これ以降では、left< right*/
52    inc=left;/*左端に設定*/
53    dec=right-1;/*右端に設定。ピボッド分のあらかじめ除く*/
54
55    while(TRUE)
56      {
57        while(TRUE)
58          {
59            /*左側で、ピボットより小さい分をスキップ。*/
60            weight();
61            if(Data[inc]>pivot || inc>=dec)
62              {
63                break;
64              }
65            inc++;
66          }
67
68        while(TRUE)
69          {
70            /*右側で、ピボットより大きい分をスキップ。*/
71            weight();
```

```
72              if(Data[dec]<pivot || dec<=inc)
73                {
74                  break;
75                }
76              dec--;
77            }
78
79          /*inc と dec が交差したら終了。*/
80          if(inc>=dec)
81            {
82              break;
83            }
84          /*ピボットより大きい要素とピボットより小さい要素の交換*/
85          swap(&Data[inc],&Data[dec]);
86        }
87      /*ピボットを正位置に設定*/
88      swap(&Data[inc],&Data[right]);
89      return inc;
90    }
```

3.5　マージソート

マージソートの実装を、リスト 19 に示す。

リスト 19:merge_sort.c

```
1   /*マージソート関数*/
2   #include <stdio.h>
3   #include "sort.h"
4
5   /*プロトタイプ宣言*/
6   void m_sort(int left,int right);/*マージソート本体*/
7   void merge(int left,int mid,int right);/*併合 (マージ)*/
8   void write_work(int left,int right);/*データを配列 Work へ退避する*/
9
10  /*作業用の配列*/
11  double Work[N];/*データの一次退避用*/
```

```
12
13  /*****************************関数定義***********************************/
14  /*m_sort(0,N-1)。他の形式に合わせるための関数*/
15  void merge_sort()
16  {
17    m_sort(0,N-1);
18    return;
19  }
20
21  /* マージソート本体。再帰を用いる。*/
22  void m_sort(int left, int right)
23  {
24    int mid;/*中間を蓄える変数*/
25
26    if(left>=right)
27      {
28        return;
29      }
30    else
31      {
32        mid=(left+right)/2;
33
34        m_sort(left,mid);
35        m_sort(mid+1,right);
36
37        write_work(left,right);
38
39        merge(left,mid,right);
40        return;
41      }
42  }
43
44  /*
45  Work[left]-Work[mid] の内容と、
46  Work[mid]-Work[right] の内容をマージして、
47  Data[left]-Data[right] をソート状態する関数
48  */
49  void merge(int left,int mid,int right)
50  {
```

```
51      int l=left;  /*配列 Work の左部分を走査する。 left<=l<=mid*/
52      int r=mid+1;  /*配列 Work の右部分を走査する。mid+1<=r<=right*/
53
54      int i=left; /*配列 Data を走査する。 left<=i<=right*/
55
56      for(i=left;i<=right;i++)
57        {
58          if(Work[l]<=Work[r] && l<=mid)
59            {
60              /*左ほうが小さい場合*/
61              Data[i]=Work[l];
62              l++;
63            }
64          else if(Work[r]<Work[l] && r<=right)
65            {
66              /*右ほうが小さい場合*/
67              Data[i]=Work[r];
68              r++;
69            }
70          else if(l>mid)
71            {
72              /*左が尽きたとき*/
73              Data[i]=Work[r];
74              r++;
75            }
76          else if(r>right)
77            {
78              /*右が尽きたとき*/
79              Data[i]=Work[l];
80              l++;
81            }
82        }
83      return;
84    }
85
86    /*
87    Data[left]-Data[right] を
88    Work[left]-Work[right] へ書き出す関数。
89    */
```

```
90  void write_work(int left,int right)
91  {
92    int i;
93
94    for(i=left;i<=right;i++)
95      {
96        Work[i]=Data[i];
97      }
98    return;
99  }
```

3.6 ヒープソート

ヒープソートの実装を、リスト 20 に示す。

リスト 20:heap_sort.c

```
1   /*ヒープソート関数*/
2   #include <stdio.h>
3   #include "sort.h"
4
5   /*マクロの追加*/
6   #define ROOT 0
7
8   /*プロトタイプ宣言*/
9   void make_heap();/*ヒープの初期化*/
10  int insert_heap(double data);/*挿入。保持しているデータ数を返す。*/
11  double get_min();/*最小値を取り出す。*/
12  void up_correct(int node);/*上方への修正*/
13  void down_correct(int node);/*下方への修正*/
14
15  /*グローバル変数*/
16  double Heap[N+1];/*ヒープ*/
17  int H_num;/*ヒープ中のデータ数。Heap[0]-Heap[H_num-1]まで利用中*/
18
19  /* ヒープソート本体。データ構造ヒープを用いる。*/
20  void heap_sort()
```

```
21  {
22    int i;
23    make_heap();
24
25    /*ヒープへのデータの挿入*/
26    for(i=0;i<N;i++)
27      {
28        insert_heap(Data[i]);
29      }
30
31    /*ヒープからのデータの取り出し*/
32    for(i=0;i<N;i++)
33      {
34        Data[i]=get_min();
35      }
36    return;
37  }
38
39  /*ヒープの初期化*/
40  void make_heap()
41  {
42    int i;
43    H_num=0;
44    for(i=0;i<N;i++)
45      {
46        Heap[i]=0.0;
47      }
48    return;
49  }
50
51  /*挿入。保持しているデータ数を返す。*/
52  int insert_heap(double data)
53  {
54    /*オーバーフロー*/
55    if(H_num>=N)
56      {
57        printf("Over flow.\n");
58        printf("%f not Inserted to Heap\n",data);
59        return -1;/*異常終了*/
```

```
60        }
61
62      /*挿入可能*/
63      H_num++;/*データ数を増加*/
64      Heap[H_num-1]=data;/*k 番目のデータは、Heap[k-1] にまず保存*/
65
66      up_correct(H_num-1);
67      return H_num;
68    }
69
70    /*最小値を取り出す。*/
71    double get_min()
72    {
73      double min;/*最小値*/
74      int left;
75
76      min=Heap[ROOT];/*根が最小値*/
77
78      Heap[ROOT]=Heap[H_num-1];
79      H_num--;/*データ数を減少*/
80
81      down_correct(ROOT);
82
83      return min;
84    }
85
86    /*上方への修正*/
87    void up_correct(int node)
88    {
89      int parent;/*親*/
90      int child; /*子*/
91
92      child=node;
93      while(child>=ROOT)
94        {
95          weight();
96          parent=(child-1)/2;
97
98          if(Heap[parent]>Heap[child])
```

```
 99          {
100            swap(&Heap[parent],&Heap[child]);
101            child=parent;
102          }
103        else
104          {
105            break;
106          }
107      }
108    return;
109  }
110
111  /*下方への修正*/
112  void down_correct(int node)
113  {
114    int left;/*左の子*/
115    int right;/*右の子*/
116    int parent;/*親*/
117
118    parent=node;
119    left=parent*2+1;
120    right=parent*2+2;
121
122    while(left<H_num)
123      {
124        weight();
125        /*子供があるとき*/
126        if(right >= H_num || Heap[left]<=Heap[right])
127          {
128            /*右の子が無いか、左の子が小さいとき。*/
129            if(Heap[parent]<=Heap[left])
130              {
131                /*親が一番小さいとき。(正当化終了)*/
132                break;
133              }
134            else
135              {
136                /*左子供が一番小さいので、親と左を交換*/
137                swap(&Heap[parent],&Heap[left]);
```

```
138                 parent=left;/*次の親を設定*/
139               }
140           }
141         else
142           {
143             /*右の子が小さいとき*/
144             if(Heap[parent]<=Heap[right])
145               {
146                 break;
147               }
148             else
149               {
150                 swap(&Heap[parent],&Heap[right]);
151                 parent=right;
152               }
153           }
154         /*新たな子を設定*/
155         left=parent*2+1;
156         right=parent*2+2;
157       }
158     return;
159   }
```

レポート 課題 S3

提出締切:2004/07/01:(Fri.)
提出先:講義直前に提出するか、GI511 レポート提出箱に提出する。

S3-R1 各種ソートの実行時間を計測し、その性能を比較考察せよ。

S3-R2 クイックソートに対して、最悪のデータ系列を生成する関数 `make_worst_data()` を作成せよ。また、最悪のデータ系列に対するクイックソートの動作と、ランダムなデータ系列に対するクイックソートの動作を比較考察せよ。

S3-R3 マージソートに対しては、関数 `weight()` が埋め込まれていない。マージソートに対しても、他の関数と同様な評価が行なえるように、関数 `weight()` を適切な部分に埋め込め。レポートは、`weight()` を埋め込んだソース `merge_sort.c` を提出すること。

S3-R4 提示しているヒープソートにおいて、データ構造用の記憶量 (配列 `Heap[]`) は、データ保持用の配列 (配列 `Data[]`) とは別に用意してあった。しかし、元のデータが保存してある記憶領域 (配列 `Data[]`) だけでも、ヒープソートは実現できる。データ構造として配列 `Heap[]` を用いずに、配列 `Data[]`) だけを用いてヒープソートを行なうようなプログラムを作成せよ。

ヒント：次の方針に従うと良い。

1. ヒープの構成を最大値が根に保存されるように変更する。
2. 配列 `Data[]` の前半をヒープとして用い、配列 `Data[]` の後半は大きい方の値がソート済みの状態で保持されるようにする。
3. 配列 `Data[]` を一たんすべてヒープの状態にして、その後で大きい方から順に配列 `Data[]` をソート済みの状態にする。

第4章　サーチ

サーチで用いられる各種宣言類をまとめたヘッダファイルを、リスト 21 に示す。

リスト 21:search.h

```
1   /*サーチ関連のヘッダ*/
2   /*マクロ*/
3   #define TRUE 1   /*真*/
4   #define FALSE 0   /*偽*/
5
6   #define NODATA -1  /*データが無いことを表わす*/
7
8   #define  N  1000  /*データ数*/
9   #define  W  100000 /*重み*/
10
11  /*関数のプロトタイプ宣言*/
12  void weight();/*1 ステップを遅くする*/
13  void make_data();/* 配列 Data の値を設定する。*/
14  void print_data();/* 配列 Data の値を表示する。*/
15  double make_key();/*Data に含まれるキーを生成する。*/
16  double rand_key();/*ランダムな実数 (0,1) をキーとして生成する。*/
17  void print_position(double key,int pos);/*探索結果 (位置) を表示する。*/
18
19  /*ソート関連*/
20  void merge_sort();/*マージソート*/
21  void swap(double *a,double *b);/*交換*/
22
23  /*探索関数*/
24  int linear_search(double key);/*線形探索*/
25  int loop_b_search(double key);/*2 分探索 (繰り返し版)*/
26  int rec_b_search(double key);/*2 分探索 (再帰版)*/
27
28  /*データを蓄えるグローバル変数*/
```

```
29  double Data[N];
```

リスト 22:search_util.c

```
1   /*サーチ関連の関数定義*/
2   #include<stdio.h>
3   #include<stdlib.h>
4   #include"search.h"
5
6   /*1 ステップを遅くする関数*/
7   void weight()
8   {
9     int i;
10    for(i=0;i<W;i++)
11      {
12        printf("");
13      }
14  }
15
16  /*
17    配列 Data の値を設定する関数
18   各値は、(0,1) の実数
19  */
20  void make_data()
21  {
22    int i;
23    srand(time(NULL));/*乱数系列の初期化*/
24
25    for(i=0;i<N;i++)
26      {
27        Data[i]=(double)rand()/(RAND_MAX+1.0);
28      }
29    return;
30  }
31
32  /* 配列 Data の中身を表示する関数*/
33  void print_data()
```

```
34  {
35    int i;
36    for(i=0;i<N;i++)
37      {
38        printf("%12.8f",Data[i]);
39        if(i%5 ==4)
40          {
41            printf("\n");
42          }
43      }
44    printf("\n");
45    return;
46  }
47
48  /*
49  Data[0] から Data[N-1] までの一つの実数
50  をランダムに選ぶ
51  */
52  double make_key()
53  {
54    int key_index;
55
56    key_index =rand() % N;
57    return Data[key_index];
58  }
59
60  /*
61  (0,1) の実数をランダムに生成
62  */
63  double rand_key()
64  {
65    double key;
66
67    key =(double)rand()/(RAND_MAX+1.0);
68    return key;
69  }
70
71  /*
72  探索結果 (位置) を表示する関数
```

```
73  */
74  void print_position(double key,int pos)
75  {
76    if(pos==NODATA)
77      {
78        printf("%12.8f はData[]に有りません。\n",key);
79      }
80    else if (0<= pos && pos <N)
81      {
82        printf("%12.8f は、Data[%d]にあります。\n",key,pos);
83      }
84    else
85      {
86        printf("ここは実行されない。");
87      }
88
89    return;
90  }
91
92  /*変数aと変数bの中身を交換する関数。
93    swap(&a,&b)として呼びだす。*/
94  void swap(double *a,double *b)
95  {
96    double tmp;
97    tmp=*a;
98    *a=*b;
99    *b=tmp;
100
101   return;
102 }
```

main関数を持つソースコードをリスト23に示す。サーチの各プログラムは、すべてこのプログラムにリンクして利用することにする。

リスト 23:test_seach.c

```
1   /*サーチをテストするプログラム*/
2   /*#include <time.h>*/
```

```
3   #include <stdio.h>
4   #include "search.h"
5
6   int main()
7   {
8     double key;
9     int  pos;
10
11    /*****************データの生成、ソート、表示*********/
12    make_data();
13    merge_sort();
14    print_data();
15
16    /************************キーの生成*****************/
17    /*key=make_key();*/       /*存在するキーの生成*/
18    key=rand_key();  /*存在しないキーの生成*/
19
20    /****************探索*****************************/
21
22    /*線形探索*/
23    pos=linear_search(key);
24    print_position(key,pos);
25
26    /*2 分探索 (繰り返し版)*/
27    pos=loop_b_search(key);
28    print_position(key,pos);
29
30
31    /*2 分探索 (再帰版)*/
32    pos=rec_b_search(key);
33    print_position(key,pos);
34
35    return 0;
36  }
```

分割されたソースファイルが増えた場合には、必要なぶんだけオブジェクトコード名を並らべれば良い。本章で最後に得られる `Makefile` は次のようになるはずである。リスト

24:`Makefile`

```
1  CC=gcc
2  objects = test_search.o search_util.o merge_sort.o \
3          linear_search.o loop_b_search.o rec_b_search.o
4
5  all:test_search
6  test_search:$(objects)
7  $(objects):search.h
```

4.1 線形探索

線形探索の実装を、リスト 25 に示す。

リスト 25:linear_search.c

```
1  #include<stdio.h>
2  #include "search.h"
3
4  int linear_search(double key)
5  {
6    int i;
7
8    for(i=0;i<N;i++)
9      {
10       weight();
11       if(Data[i]==key)
12         {
13           return i;
14         }
15     }
16
17   return NODATA;
18 }
19
```

4.2 2 分探索 (繰り返し版)

繰り返しを用いた 2 分探索の実装を、リスト 26 に示す。

リスト 26:loop_b_search.c

```
1  /*繰り返しを用いて 2 分探索するプログラム*/
2  #include<stdio.h>
3  #include "search.h"
4
5  /*
6    繰り返しを用いて 2 分探索する関数
7    引数:
8    key  キー
9    戻り値 :
10   キーが存在しないとき、NODATA=-1
11   キーが存在するとき、キーのある配列の添字
12 */
13 int loop_b_search(double key)
14 {
15   int left=0;/*探索範囲の左端*/
16   int right=N-1;/*探索範囲の右端*/
17   int mid=-1;/*探索範囲の中間*/
18
19   while(left<=right)
20     {
21       weight();
22
23       mid=(left+right)/2;
24
25       if(Data[mid]==key)
26         {
27           printf("発見\n");
28           return mid;
29         }
30       else if(key<Data[mid])
31         {
32           printf("小さい方にある。\n");
33           right=mid-1;
```

```
34          }
35        else if( Data[mid]<key)
36          {
37            printf("大きい方にある。\n");
38            left=mid+1;
39          }
40      }
41
42    return NODATA;
43  }
```

4.3 2分探索(再帰版)

再帰を用いた2分探索の実装を、リスト27に示す。

リスト 27:rec_b_search.c

```
1   #include<stdio.h>
2   #include "search.h"
3
4   /*2分探索本体のプロトタイプ宣言*/
5   int b_search(double key,int left,int right);
6
7   /*探索の形式を合わせる関数*/
8   int rec_b_search(double key)
9   {
10    return b_search(key,0,N-1);
11  }
12
13  /*
14   再帰を用いて2分探索する関数
15   引数：
16   key  キー
17   left 探索する配列の左端の添字
18   right 探索する配列の右端の添字
19   戻り値：
20   キーが存在しないとき、NODATA=-1
```

```
21    キーが存在しするとき、配列の添字
22  */
23  int b_search(double key,int left,int right)
24  {
25    int mid;/*探索範囲の中間の添字*/
26
27    weight();
28    if(left>right)
29      {
30        /*基礎*/
31        return NODATA;
32      }
33    else
34      {
35        /*帰納*/
36        mid=(left+right)/2;
37
38        if(Data[mid]==key)
39          {
40            printf("発見\n");
41            return mid;
42          }
43        else if(key<Data[mid])
44          {
45            printf("小さい方にある。\n");
46            return b_search(key,left,mid-1);
47          }
48        else if (Data[mid]<key)
49          {
50            printf("大きい方にある。\n");
51            return b_search(key,mid+1,right);
52          }
53      }
54  }
```

4.4 ハッシュ

ハッシュを用いた探索をリスト 28 に示す。

リスト 28:testhash.c

```
1  /*内部ハッシュ法による探索*/
2  #include <stdio.h>
3  #include <string.h>
4
5  #define MEMBER 100  /*人数*/
6  #define NAME   10   /*名前の最大長*/
7
8  #define TRUE 1  /*真*/
9  #define FALSE 0 /*偽*/
10
11 #define NODATA (-1) /*空を表す*/
12
13 #define W 100000   /*重みの最大値*/
14
15 char Name[MEMBER][NAME]; /*名前を保存する配列*/
16
17 void init();   /*Name を初期化する関数*/
18 void input();  /*Name に名前を標準入力から読み込む関数*/
19 void print();  /*Name を表示する関数*/
20 int hash(char *name);  /*名前のハッシュ値を求める関数*/
21 int search(char *name); /*名前を探す関数*/
22 void weight();  /*1 ステップを遅くする関数。マクロ W で制御*/
23
24
25 int main()
26 {
27   char name[NAME];
28   int  position=NODATA;
29   /*入力*/
30   init();
31   input();
32   print();
33
```

```
34      /*探索*/
35      printf("探索する名前:");
36      scanf("%s",name);
37
38      position=search(name);
39      if(position==NODATA)
40        {
41          printf("%s はありません。\n",name);
42        }
43      else
44        {
45          printf("%s は Name[%d] にあります。\n",name,position);
46        }
47
48      return 0;
49    }
50
51    /*
52    Name を初期化する関数
53    引数:なし
54    戻り値:void
55    */
56    void init()
57    {
58      int i;
59      for(i=0;i<MEMBER;i++)
60        {
61          Name[i][0]='\0';
62        }
63
64      return;
65    }
66
67    /*
68    Name を表示する関数
69    引数:なし
70    戻り値:void
71    */
72    void print()
```

```
73  {
74    int i;
75    for(i=0;i<MEMBER;i++)
76      {
77        printf("%s \n",Name[i]);
78      }
79
80    return;
81  }
82
83  /*
84  ハッシュ関数
85  引数:名前を表わす文字列
86  戻り値:ハッシュ値
87  */
88  int hash(char *name)
89  {
90    int i=0;
91    int sum=0;
92
93    if(name[0]=='\0')
94      {
95        return NODATA;
96      }
97    else
98      {
99        while(name[i]!='\0')
100         {
101           sum+=(int)name[i];
102           i++;
103         }
104       return (sum % MEMBER);
105     }
106 }
107
108 /*
109 ファイルから、名前を読み込む関数
110 入力ファイル名:testhash.in
111 引数:なし
```

```
112  戻り値:void
113  */
114  void input()
115  {
116    int i=0;
117    int h=NODATA;
118    char name[NAME];
119    FILE  *fin;/*入力ファイルを指定するファイルポインタ*/
120
121    fin=fopen("testhash.in","r");/*入力ファイルを(読み出し指定で)開く*/
122
123    while(fscanf(fin,"%s",name)!=EOF)
124      {
125        h=hash(name);
126        while(Name[h][0]!='\0')
127          {
128            weight();
129            h=(h+1)%MEMBER;
130          }
131        strcpy(Name[h],name);/*nameを配列Nameへ挿入*/
132      }
133
134    fclose(fin);/*入力ファイルを閉じる。*/
135
136    return;
137  }
138
139  /*
140  名前を探索する関数
141  引数:名前を表わす文字列
142  戻り値:配列Nameに存在する添字
143  */
144  int search(char *name)
145  {
146    int h=NODATA;
147
148    h=hash(name);
149    while(TRUE)
150      {
```

```
151         if(Name[h][0]=='\0')
152           {
153             return NODATA;
154           }
155
156         if(strcmp(Name[h],name)==0) /*配列 Name に name を見つける*/
157           {
158             break;
159           }
160         weight();
161         h=(h+1)%MEMBER;
162       }
163
164     return h;
165   }
166
167   /*
168   1 ステップを遅くする関数。マクロ W で制御
169   引数:なし
170   戻り値:void
171   */
172   void weight()
173   {
174     int i;
175     for(i=0;i<W;i++)
176       {
177         printf("");
178       }
179
180     return;
181   }
```

レポート 課題 S4

提出:2004/07/23:(Fri.) 提出先:GI511 レポート提出箱

S4.R1 線形探索、2 分探索 (繰り返し版)、2 分探索 (再帰版) の各探索法について、実行時間を計測しそれらの性能を比較考察せよ。

S4.R2 2 分探索木について調べて以下に答えよ。

(1) n 個のデータに対して 2 分探索木を作成するとき、作成に必要な最悪計算量と平均計算量を示せ。

(2) n 個のデータを保持している 2 分探索木に対して探索を行なったとき、探索に必要となる最悪時間計算量と平均時間計算量を示せ。

S4.R3 ハッシュ法において、衝突の回数を計測できるようにプログラムを変更せよ。また、配列の利用効率と衝突の関係を考察せよ。なお、参考の入力データを、次のファイルに用意したので、コピーして利用して下さい。`/home/student/submit/S4/R3/testhash.in`