I117（6）整列

知念

北陸先端科学技術大学院大学 情報科学研究科
School of Information Science,
Japan Advanced Institute of Science and Technology

おことわり

- 整列（ソート）各種技術はアルゴリズムの講義で
- ここでは軽く、ソートの利用方法を紹介する
- メインメモリ上のソート関数 qsort
  - 多く場合 qsort で事足りる
  - 例外
    - 高速性が要求される => 関数の形態を変更
    - 特殊なデータ内容 => アルゴリズムを変更
    - 配列では格納されていない

※ 外部メモリを使うソートは扱わない

qsort

- クイックソート（quick sort）の汎用的実装
- 多くの処理系で利用可能

```
void qsort(void *base, size_t nel,
    size_t width,
    int (*compar)(const void *,
                  const void *));
```

- 引数
  - 場所、要素数、要素サイズ、比較関数
  - 間違えやすいので注意

比較関数

```
int (*compar)(cons void*, const void*);
```

qsort は比較関数で整列順を決める
返り値に前提をおく

- $>$0 大きい
- $==$0 同じ
- $<$0 小さい

比較の詳細は比較関数の実装任せ

比較関数: int の比較

※ Solaris の man から引用

```
static int intcompare(const void *p1,
    const void *p2)
{
    int i = *((int *)p1);
    int j = *((int *)p2);
    if (i > j)  return (1);
    if (i < j)  return (-1);
    return (0);
}
```

比較関数: int の比較 *(cont.)*

実はこの程度で十分

```
static int intcompare(const void *p1,
    const void *p2)
{
    int i = *((int *)p1);
    int j = *((int *)p2);
    return i-j;
}
```

呼び出し例

```
int ar[] = { 41, 3, 92, 7, 13, 10};
int i;

qsort((void *)ar,
    sizeof(ar)/sizeof(int),
    sizeof(int), intcompare);
for (i = 0; i < sizeof(ar)/sizeof(int);
        i++) {
    printf("%d ", ar[i]);
}
printf("\n");
```

整数整列

結果 — 昇順↗

```
% ./a.out
3 7 10 13 41 92
```

比較関数の引き算を j - i にすると降順↘

```
% ./a.out
92 41 13 10 7 3
```

比較関数一般形

整列対象のデータ型を Utype とする

```
int cmp(const void*x, const void*y) {
    Utype xp = *(Utype*)x;
    Utype yp = *(Utype*)y;
    return <compare xp and yp>;  }
```

Utype が int なら

```
    int xp = *(int*)x;
```

Utype が char* なら

```
    char *xp = *(char**)x;
```

比較関数一般形 — 要素が基本型の場合

Utype	`Utype`	`xp = *(Utype *)x;`

整数 int	`int`	`xp = *(int *)x;`
文字 char	`char`	`xp = *(char *)x;`
文字列 char*	`char*`	`xp = *(char* *)x;`

比較関数一般形 — 要素が構造体の場合

要素が構造体でそのメンバを比較、という場面は多い
次のような構造体からなる型 UST を考えると

```
typedef struct {
    char *key;
    void *cont;
} UST;
```

Utype	`Utype`	`xp = *(Utype *)x;`
構造体 UST	`UST`	`xp = *(UST *)x;`

比較関数一般形 — 要素が構造体の場合 *(cont.)*

比較に用いる文字列 key の取り出しは

```
    UST xp = *(UST*)x;
    char *xkey = xp.key;
```

xp をポインタで表現すると

```
    UST *xp = (UST*)x;
    char *xkey = xp->key;
```

xp を使わず一気に表現すると

```
    char *xkey = ((UST*)x)->key;
```

比較関数一般形 — 要素が構造体の場合 *(cont.)*

同様に取り出した ykey と比較する

```
    char *ykey = ((UST*)y)->key;
    return strcmp(xkey, ykey);
```

※ 構造体を扱う整列は後述

文字列整列

文字列の大小比較に関数 strcmp を使う

```
int cmp(const void *x, const void *y)
{
    char *xp = *(char**)x;
    char *yp = *(char**)y;
    return strcmp(xp, yp);
}
```

char** が出て来ても驚かない

呼び出し側

```
char *ar[] = { "41", "3", "92",
          "7", "13", "10"};
int i;
qsort((void *)ar,
    sizeof(ar)/sizeof(char*),
    sizeof(char*), cmp);
for (i = 0;
  i < sizeof(ar)/sizeof(char*); i++) {
    printf("%s ", ar[i]);
}
printf("\n");
```

文字列整列 *(cont.)*

結果

```
10 13 3 41 7 92
```

文字列なので、”13”＜”3” ”41”＜”7”

- strcmp の挙動

1) 先頭から文字の大小比較
2) 短い文字列が小さい返り値

文字列整列 *(cont.)*

アルファベットにすると

```
char *ar[] = { "openlog", "syslog",
        "closelog", "syslogmask"};
```

結果

```
closelog openlog syslog syslogmask
```

降順は -strcmp(xp,yp); か strcmp(yp,xp); に

```
syslogmask syslog openlog closelog
```

文字列整列 *(cont.)*

空白を含む長い文字列

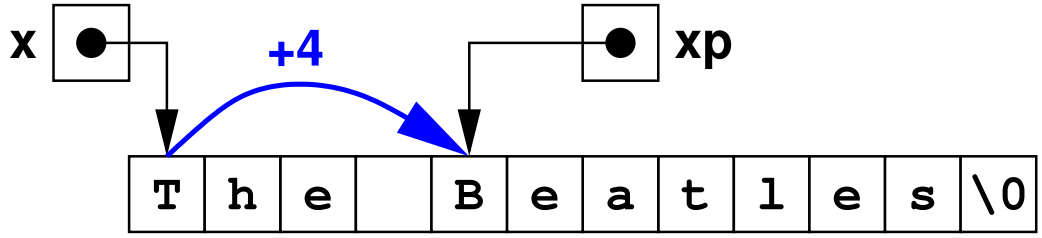
```
char *ar[] = {"The Beatles", "U2",
    "Norah Jones", "Johnny Cash",};
```

結果 — 1項目ずつ改行してある

```
Johnny Cash
Norah Jones
The Beatles
U2
```

データ内容に合わせた変更 — The を除いて比較

```
int cmp(const void *x, const void *y) {
    char *xp, *yp;
    xp = *(char**)x;
    if(strncasecmp(xp,"the ",4)==0) {
        xp += 4; }
    yp = *(char**)y;
    if(strncasecmp(yp,"the ",4)==0){
        yp += 4; }
    return strcasecmp(xp, yp);
}
```



文字列整列 *(cont.)*

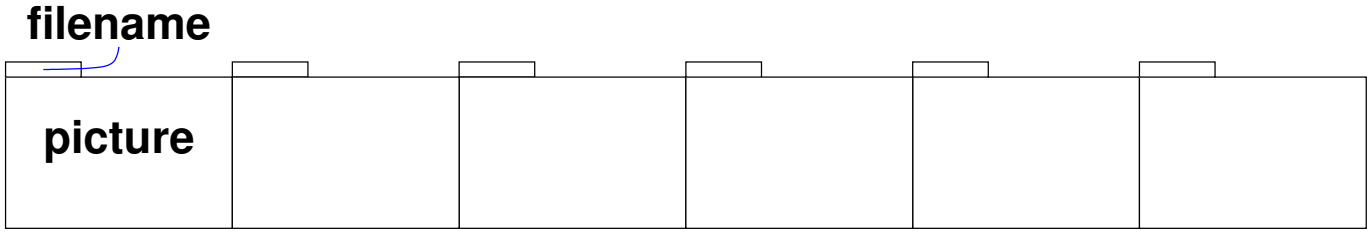
- The だけでなく空白も含めて比較
  - There, Them 等の単語を誤処理しないため
- 大文字小文字混在なので strcasecmp を使う

結果

```
The Beatles
Johnny Cash
Norah Jones
U2
```

参照配列によるソート

- 対象データ群が大きい
  - ソートに伴うコピーに時間がかかる
    数10MB/件 の画像を名前でソートしたり



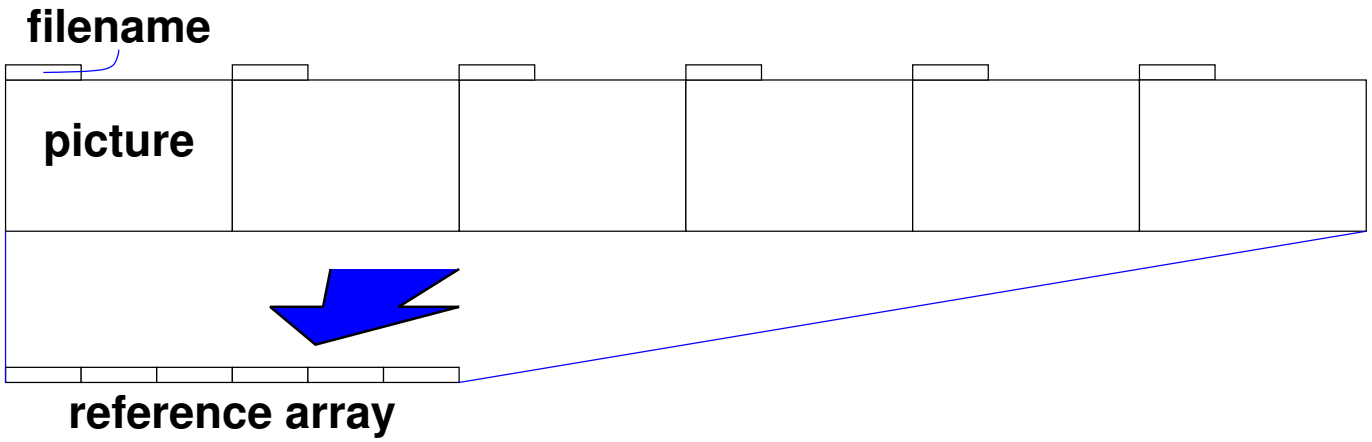
sizeof(filenames)<<sizeof(pictures)

- 対象データが線形に格納されていない
  - qsort が扱えない

=> ソートのために配列を別に設ける

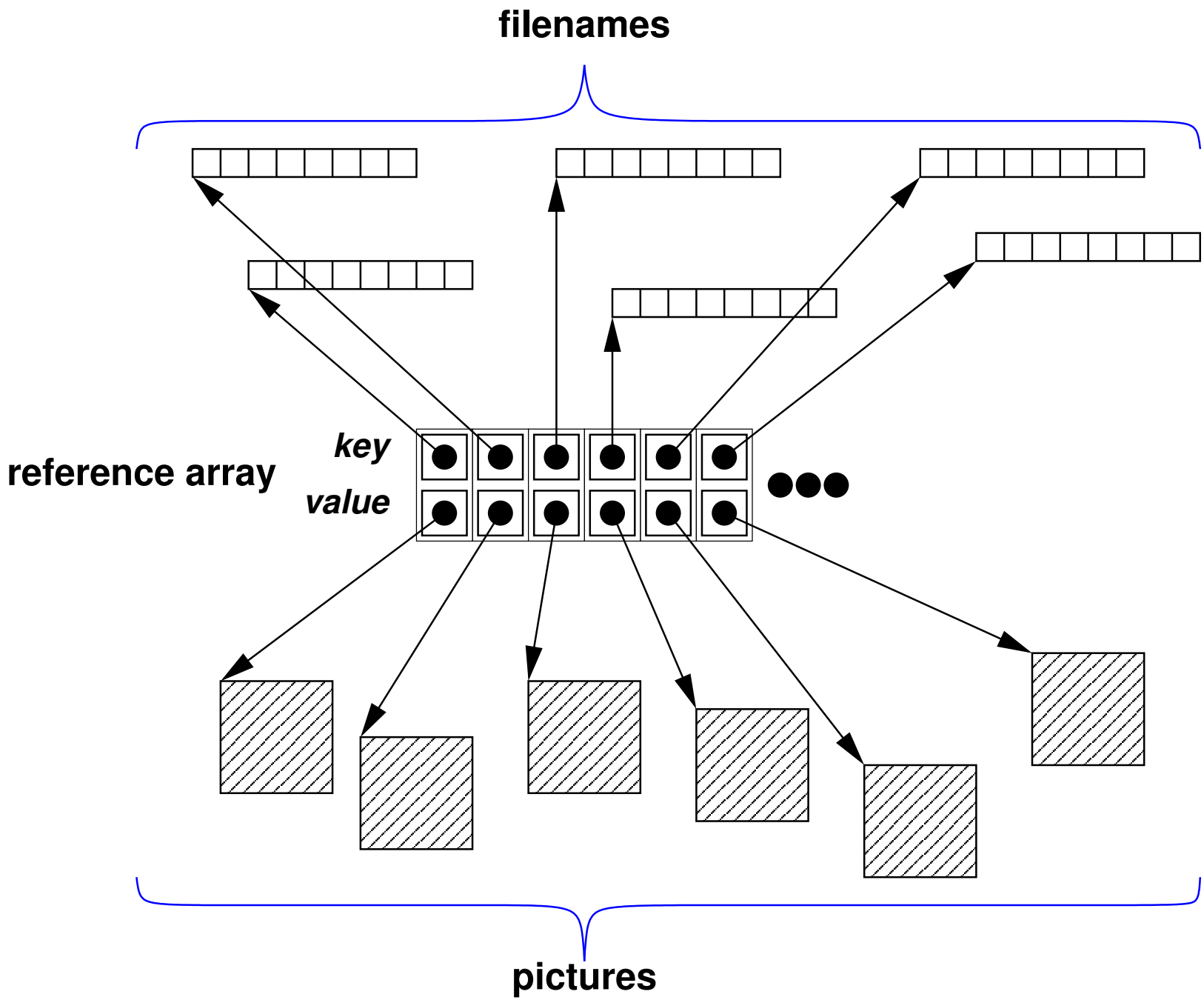
参照配列によるソート *(cont.)*

大略



ソート結果を使うために元データへの参照も設ける

```
typedef struct {
    char *key;
    void *value;
} cell;
```


filenames
reference array
key
value
pictures

参照配列によるソート *(cont.)*

こうすると...

- ソート処理中のコピーは参照配列だけ
  - 小さな領域（8バイト）に押し込められる
- 参照配列のソート後、参照配列をたぐるだけで対象データがソートされたことになる。

ちなみに、参照メンバ value の型を void* にしたのは任意のポインタを扱うため

画像は大変なので、ここでは文字列でサンプルを作る

```
char* pickstr2key(void *src) {
    char *dst, *p = src, *q;
    dst = (char*)malloc(strlen(src)+1);
    if(strncasecmp(src, "the ", 4)==0) {
        p += 4;
    }
    q = dst;
    while(*p) {
        *q++ = toupper(*p++);
    }
    *q = '\0';
    return dst;  }
```

conv は pickstr2key を使う

```
int str2cell(cell dst[], void *src[],
    int nlen, char *(conv)(void*))
{
    int i;
    for(i=0;i<nlen;i++) {
        dst[i].key = conv(src[i]);
        dst[i].value = src[i];
    }
    return 0;
}
```

参照配列によるソート *(cont.)*

呼び出し側

- 配列ar の長さn は sizeof(ar)/sizeof(ar[0]) で算出
- 参照配列 refar の長さは sizeof(cell)*n

```
char *ar[] = {"The Beatles", "U2",
    "Norah Jones", "Johnny Cash",};
cell *refar;
int n = sizeof(ar)/sizeof(ar[0]);
refar = (cell*)malloc(sizeof(cell)*n);
str2cell(refar, (void*)ar, n,
         pickstr2key);
```

参照配列によるソート *(cont.)*

あとは qsort を適用して、表示する

```
qsort(refar, n, sizeof(refar[0]),
          cellkeycmp);
for(i=0;i<n;i++) {
    printf("%-16s %s\n",
          refar[i].value, refar[i].key);
}
```

- cellkeycmp は単純な strcmp で十分
- refar のソート結果から ar の内容を表示

参照配列によるソート *(cont.)*

pickstr2key で加工(”the ”除去、大文字化)しているので単純な比較で良い

```
int cellkeycmp(void const *xp,
        void const *yp)
{
    char *x; char *y;
    x = ((cell*)xp)->key;
    y = ((cell*)yp)->key;
    return strcmp(x,y);
}
```

参照配列によるソート *(cont.)*

結果

```
The Beatles          BEATLES
Johnny Cash          JOHNNY CASH
Norah Jones          NORAH JONES
U2                   U2
```

- 元データ操作なし、ソート中のコピー処理が軽い
- ソート前に加工済、キー比較の処理が軽い
- 参照配列の分だけメモリを消費（デメリット）

複キー

キーを複数持つ場合もある

```
typedef struct {
    char *key1;
    char *key2;
    char *cont;
} datatype;
```

ここでは、key1 が key2 に優先すると定める

複キーの比較関数

```
int cmp(const void *pa,
        const void *pb) {
    int c1, c2;
    datatype *a = (datatype*)pa,
             *b = (datatype*)pb;
    c1 = strcmp(a->key1, b->key1);
    c2 = strcmp(a->key2, b->key2);
    if(c1==0)
        return c2;
    return c1;
}
```

複キー *(cont.)*

拡張子を優先キーに

```
datatype data[] = {
    {"c",   "foo",      "foo.c"},
    {"awk", "bar",      "bar.awk"},
    {"c",   "bar",      "bar.c"},
    {"awk", "junk",     "junk.awk"},
    {"pl",  "junk",     "junk.pl"},
    {"c",   "dummy",    "dummy.c"},
    {"",    "Makefile", "Makefile"},
    {"",    "makefile", "makefile"}, };
```

結果 — 拡張子、ベース名の順に整列

before	after
foo.c	Makefile
bar.awk	makefile
bar.c	bar.awk
junk.awk	junk.awk
junk.pl	bar.c
dummy.c	dummy.c
Makefile	foo.c
makefile	junk.pl

複キー *(cont.)*

- 今回は 2個のキー
- より多数のキーも同様
  - メンバが増えて、比較関数が長くなる
  - 丁寧に書けばさほど困難ではない

おまけ

最近のコンパイラではメンバを指定して初期化可能
初期化不要なメンバを記述せずにすむので便利

```
datatype data[] = {
  { .key1 ="c"  , .key2 ="foo"     , .cont="foo.c"   },
  { .key1 ="awk", .key2 ="bar"     , .cont="bar.awk" },
  { .key1 ="c"  , .key2 ="bar"     , .cont="bar.c"   },
  { .key1 ="awk", .key2 ="junk"    , .cont="junk.awk"},
  { .key1 ="pl" , .key2 ="junk"    , .cont="junk.pl" },
  { .key1 ="c"  , .key2 ="dummy"   , .cont="dummy.c" },
  { .key1 =""   , .key2 ="Makefile", .cont="Makefile"},
  { .key1 =""   , .key2 ="makefile", .cont="makefile"},
};
```

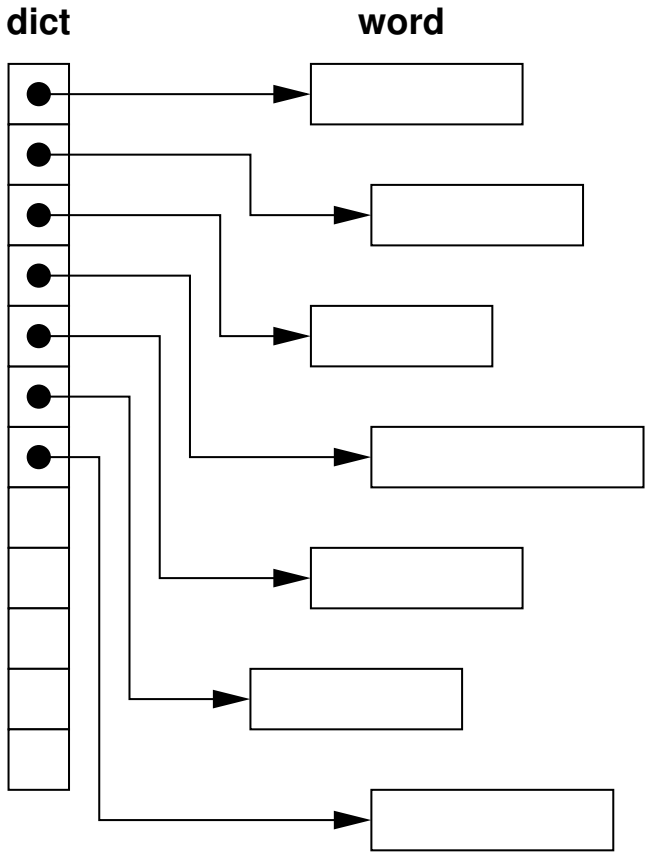
辞書

典型的な操作

- 単語列から単語一覧を作る
- 重複は省く

簡単化のため

- 配列に格納
- qsort で整列
- bsearch で探索



単語列サンプル

```
char *seq[] = {
  "man", "find", "and", "display",
  "reference", "manual", "pages",
  "man", "command", "displays",
  "information", "from", "the",
  "reference", "displays", "complete",
  "manual", "pages", "that", "you",
  "select", "by", "or", "summaries",
  "selected", "either", "by",
  "keyword", "or", "by", "the", "name",
  "of", "an", "associated", "file", };
```

辞書 *(cont.)*

配列確保

```
char **dict;
int    dictlen, dictuse;
...
dictlen = sizeof(seq)/sizeof(seq[0]);
dict    = (char**)malloc(sizeof(char*)
                         *dictlen);
dictuse = 0;
```

重複を省きながら(探索しながら)格納

```
char *ppos, *key;  int i;
for(i=0;i<sizeof(seq)/sizeof(seq[0]);
        i++) {
  key = seq[i];
  ppos = bsearch(&key, dict,
    dictuse, sizeof(dict[0]), Xstrcmp);
  if(!ppos) {
    dict[dictuse++] = strdup(seq[i]);
    qsort(dict, dictuse,
      sizeof(dict[0]), Xstrcmp);
  } }
```

bsearch

二分探索（binary search）の汎用的実装

```
void *bsearch(const void *key,
    const void *base, size_t nel,
    size_t size,
    int (*compar)(const void *,
                  const void *));
```

引数は参考とするデータ、対象探索データのアドレス、要素数、要素長、比較関数（qsort と同様）

辞書作成 結果

単純に出力する

```
    for(i=0;i<dictuse;i++){
        printf("%2d %s\n", i, dict[i]);
    }
```

辞書作成 結果 *(cont.)*

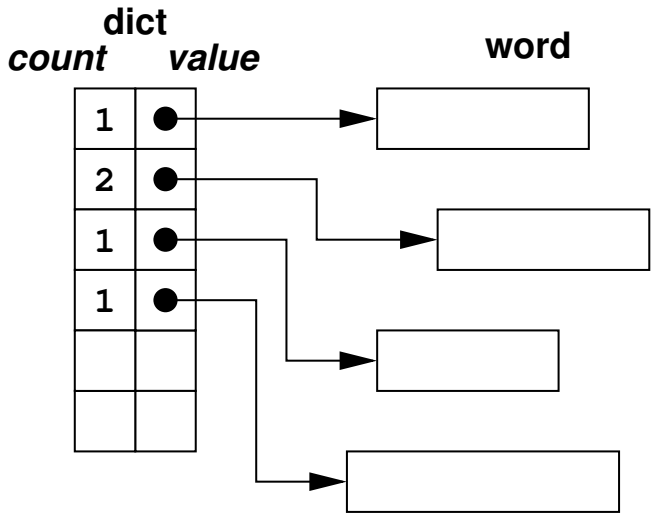
結果(長いので pr で折曲げた) のべ36単語が27単語に

```
 0 an             9 file          18 or
 1 and           10 find          19 pages
 2 associated    11 from          20 reference
 3 by            12 information   21 select
 4 command       13 keyword       22 selected
 5 complete      14 man           23 summaries
 6 display       15 manual        24 that
 7 displays      16 name          25 the
 8 either        17 of            26 you
```

辞書 頻度計上

単語の出現頻度を計上する — 型dict_t を作成

```
typedef struct {
    char *value;
    int   count;
} dict_t ;
int dictcmp(const void *ap,
        const void *bp) {
    char *a = ((dict_t*)ap)->value;
    cahr *b = ((dict_t*)bp)->value;
    return strcmp(a,b); }
```



辞書 頻度計上 *(cont.)*

領域確保はほぼ同じ

```
dict_t *dict;
int    dictlen, dictuse;

dictlen = sizeof(seq)/sizeof(seq[0]);
dict = (dict_t*)malloc(sizeof(dict_t)
                       *dictlen);
dictuse = 0;
```

ループ内で重複した際にカウンタを増やす

```
dict_t *ppos, ref;
```

ループ内部

```
ref.value = seq[i];
ppos = bsearch(&ref, dict, dictuse,
         sizeof(dict[0]), dictcmp);
if(!ppos) { /* not found */
  dict[dictuse].value = strdup(seq[i]);
  dict[dictuse].count = 1;  dictuse++;
  qsort(dict, dictuse, sizeof(dict[0]),
    dictcmp);  }
else { /* found */
  ppos->count++;  }
```

出力

```
printf("%2d %d %s\n", i,
  dict[i].count, dict[i].value);
```

結果

```
 0 1 an              9 1 file          18 2 or
 1 1 and            10 1 find          19 2 pages
 2 1 associated     11 1 from          20 2 reference
 3 3 by             12 1 information   21 1 select
 4 1 command        13 1 keyword       22 1 selected
 5 1 complete       14 2 man           23 1 summaries
 6 1 display        15 2 manual        24 1 that
 7 2 displays       16 1 name          25 2 the
 8 1 either         17 1 of            26 1 you
```

辞書 補足

- 配列は非常に単純な例
  - 木やオープンチェーン等を使う
  - 規模が小さな場合は配列も有効
- 毎回整列するのは無駄が多い
  - 挿入ソートで十分
- 格納方法によっては bsearch は向いていない

演習

1) 先の空白を含む長い文字列のソートを、2番目の単語をもとにソートするプログラムに変更せよ★
   - 第2単語が無い場合は空文字列 "" として扱え
2) 先の辞書のプログラムを単語 an と the を格納しないよう作り直せ★
3) 先の辞書のプログラムを qsort を使わず挿入ソートで格納するプログラムに作り直せ★★
4) 先の辞書のプログラムの構造体に文字数を格納するメンバを加え、各単語の文字数を実際に格納するプログラムを作れ★

演習 *(cont.)*

文字列を格納した片方向リストをソートする

5) 参照配列を使ってソートするプログラムを作れ★

6) それ以外の方法でソートするプログラムを作れ★★★