I117（20）リストによる辞書の実装

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

辞書の実装比較

指標

性質・制限

- ランダムアクセス
- 扱える項目数

処理時間

- 登録
- 検索
- 整列
- 削除

まだ削除はこれまで扱っていないので、後回し

配列とリストの比較

これまで実装した辞書で使った配列とリストの比較

	一般的な配列	実装した配列	リスト
ランダムアクセス	可能	可能	不可
長さの制限	厳しい	ゆるい※	ゆるい
メモリ消費の見積り	容易	容易	困難
整列	簡単	簡単	面倒

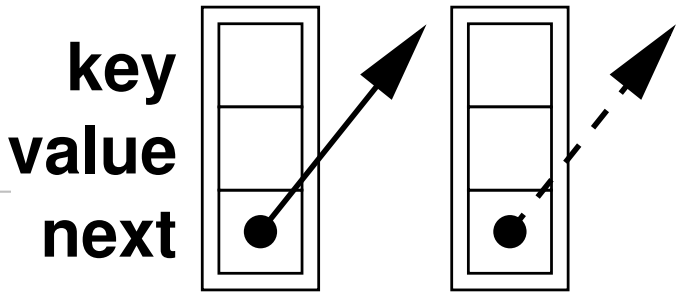
※配列の長さ制限を克服するため expand を用意した

ランダムアクセスや整列が頻繁になければ、リストによる実装も有効

リストによる辞書の実装

型定義

```
typedef struct _idict_ {
    char *key;
    int   value;
    struct _idict_ *next;
} idict_c;
```



メンバ next を使ってリストを作る

リストによる辞書の実装 *(cont.)*

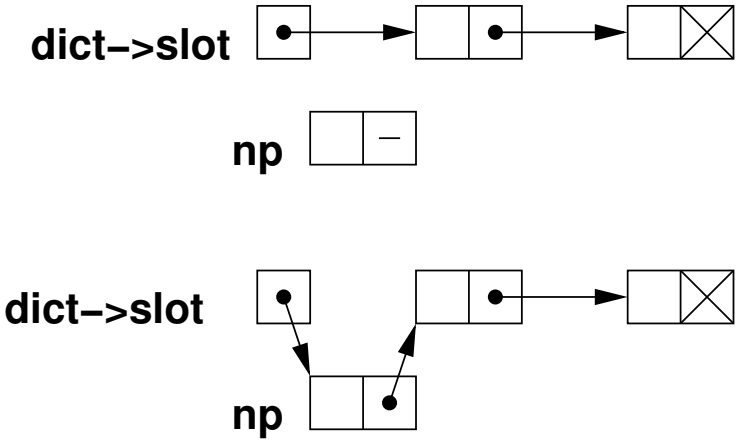
初期化ルーチンは軽く、領域拡張は不要

```
int idict_init(idict_a *dict) {
  dict->len  = 0;
  dict->use  = 0;
  dict->slot = NULL;
  return 0;
}
int idict_expand(idict_a *dict) {
  fprintf(stderr, "idict_expand: not available\n");
  return 0;
}
```

整列していないので、全て走査する

```
idict_c* idict_findpos(idict_a*dict,char*xkey){
  idict_c *ppos, *p = dict->slot;
  ppos = NULL;
  while(p) {
    if(strcmp(p->key, xkey)==0) {
      ppos = p;
      break;  }
    p = p->next;
  }
  return ppos;
}
```

```
int idict_add(idict_a*dict,char*xkey,int xvalue){
  idict_c *np;
  np = (idict_c*)malloc(sizeof(idict_c));
  if(!np) {
    fprintf(stderr, "idict_add: no memory\n");
    return -1;
  }
  np->next = dict->slot;   dict->slot = np;
  np->key = strdup(xkey);  np->value = xvalue;
  dict->use++;  dict->len++;
  return 0;
}
```



整列（瞬間的に格納分と同量メモリを消費）

```
int idict_sortbykey(idict_a *dict) {
  int i;  idict_c *ar, *p=dict->slot;
  ar = (idict_c*)malloc(
          sizeof(idict_c)*dict->use);
  for(i=0;p && i<dict->use;i++, p = p->next) {
    ar[i].key = p->key; ar[i].value = p->value; }
  qsort(ar, dict->use,
    sizeof(idict_c), idict_keycmp);
  p = dict->slot;
  for(i=0;p && i<dict->use;i++, p = p->next) {
    p->key = ar[i].key; p->value = ar[i].value;  }
  free(ar);
}
```

性能比較 (データ数 N; 単位 ms)

登録時に毎回の整列をしない配列の実装も計測

N	method	store	lookup		sort	lookup	
			10	*1k*		*10*	*1k*
1000	sortarray	437	0	1	0	0	1
	nosortarray	3	0	27	0	0	2
	list	0	0	23	1	0	24
10000	sortarray	64406	0	2	11	0	2
	nosortarray	12	4	349	13	0	2
	list	10	4	345	16	4	350

性能比較 (データ数 N; 単位 ms) *(cont.)*

整列配列: 格納は遅いが検索は早い

非整列配列: 格納は早いが検索は遅い

- リストと同程度
- 整列後なら整列配列と同程度

リスト: 格納は早いが検索が遅い

状況に応じて使い分けると良いだろう

用途

計測した範囲では...

- 格納と検索が混ざっている場合
  配列整列が良い
- 格納と検索がはっきり分かれているような場合
  整列配列は遅すぎる
  非整列配列として格納して、検索前に整列すれば良い
- 検索や整列が不要なら (順不同で保持する)
  一括登録するならリストが良い

補足

指標をかえると判断が変わるだろう

削除

- 一般的にはリストが早い
- 配列は遅いが幾らか工夫の余地がある

削除

配列なら findpos で削除箇所がみつかる

```
int idict_del(idict_a *dict, char *xkey) {
    idict_c *pos = idict_findpos(dict, xkey);
    if(pos) { pos->key[0] = '\0'; return 0; }
    return 1;
}
```

key を無効にするだけ
内容を引っ越したりすると重い処理になる

※ use に無効データが含まれる点に注意

削除 *(cont.)*

整列配列も同様に findpos を使う、そして整列

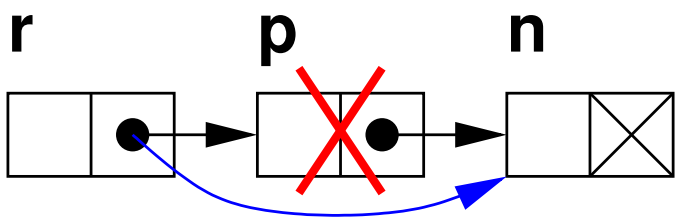
```
int idict_del(idict_a *dict, char *xkey) {
    idict_c *pos = idict_findpos(dict, xkey);
    if(pos) {
        pos->key[0] = '\0';
        qsort(dict->slot, dict->use,
            sizeof(dict->slot[0]), idict_keycmp);
        return 0;
    }
    return 1;
}
```

削除 *(cont.)*

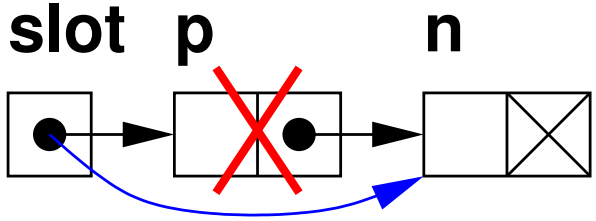
リストの場合、対象項目を指す前の項目も必要なため、findpos では不十分、自ら探索

```
int idict_del(idict_a *dict, char *xkey) {
    idict_c *p=dict->slot, *r=NULL, *n;
    while(p) {
        if(strcmp(p->key, xkey)==0) {
            break;
        }
        r = p;  p = p->next;
    }
```

```
    if(p) {
        n = p->next;
        if(r) { r->next = n;  }
        else { dict->slot = n;  }
        free(p->key);  free(p);
        return 0;
    }
    return 1;
}
```



- 先頭だけ別扱い



削除テストプログラム

```
/* test of remove items in idict */
#include <stdio.h>
#include <stdlib.h>
#include "libidict.h"
int main(){
  idict_a *idict;  int ck;

  idict = idict_new();  idict_init(idict);
  idict_add(idict,"A",1); idict_add(idict,"B",2);
  idict_add(idict,"C",3); idict_add(idict,"D",4);
  idict_add(idict,"E",5);
```

削除テストプログラム *(cont.)*

```
  printf("before remove\n"); idict_print(idict);
  ck = idict_del(idict, "A");
  if(ck) printf("ck %d\n", ck);
  ck = idict_del(idict, "C");
  if(ck) printf("ck %d\n", ck);
  ck = idict_del(idict, "E");
  if(ck) printf("ck %d\n", ck);
  printf("after remove\n"); idict_print(idict);
}
```

Ａから Ｅ の 5項目登録して、最初、最後、真中を削除

list

```
before remove
 0 E 5
 1 D 4
 2 C 3
 3 B 2
 4 A 1
after remove
 0 D 4
 1 B 2
```

array-sort

```
before remove
 0 A 1
 1 B 2
 2 C 3
 3 D 4
 4 E 5
after remove
 0  1
 1  3
 2  5
 3 B 2
 4 D 4
```

array-unsort

```
before remove
 0 A 1
 1 B 2
 2 C 3
 3 D 4
 4 E 5
after remove
 0  1
 1 B 2
 2  3
 3 D 4
 4  5
```

削除込み性能比較

N	name	store	lookup	sort	lookup	remove
		N	*1k*	*N*	*1k*	*1k*
1000	ars	438	1	0	1	413
	arns	3	27	0	0	37
	list	0	26	1	23	23
10000	ars	65213	2	11	2	12804
	arns	12	344	14	2	373
	list	9	397	16	332	340

削除後に毎回整列する分、整列配列は削除が遅い

用途再考

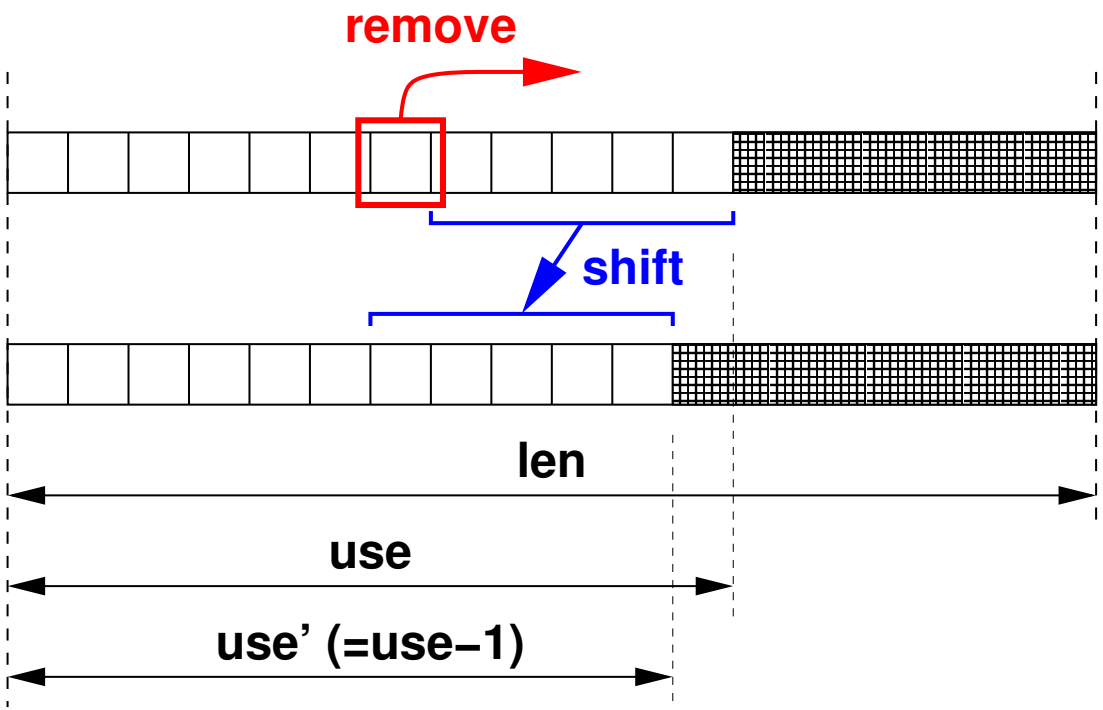
- 検索が頻繁な場合
  整列配列が有利
- 削除が頻繁な場合
  リストが有利
- 追加が頻繁な場合
  リストや非整列配列が有利

※ あくまでもこれまでの講義で出て来た範囲での検討

演習

1) リストによる辞書ライブラリを実際に作れ★
2) 非整列配列による辞書ライブラリを実際に作れ★
   - 整列配列の実装を改造
3) 自作の辞書プログラムの性能を計測せよ★★
4) 配列による辞書において、削除の際に後続項目を引越しするよう変更せよ★★
   - use の値を変更する点に注意
   - 実際のメモリ内容の転送は memcpy(), memmove() を利用すると良い

演習 *(cont.)*



演習 *(cont.)*

5) 配列とリスト以外の格納方法を用いた場合の有利・不利を検討せよ★★

- 木構造とその派生
- ハッシュとその派生
- それ以外

おまけ

50万文字列(重複無考慮)の格納、整列、検索
単位はms、有効数字は2桁程度だと思われる

method	store	sort	l10*	l1k*
list	514	1879	661	6602
array sort IDICTLEN=500000	¶>93000000	?	?	
array nosort IDICTLEN=10	403	1562	0**	6**
array nosort IDICTLEN=500000	346	1560	0**	6**

¶ 1日以上かかったので打ち切った

*lM = lookup random M strings, **lookup after sort