I117（23）ハッシュによる辞書の実装

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

ハッシュの導入

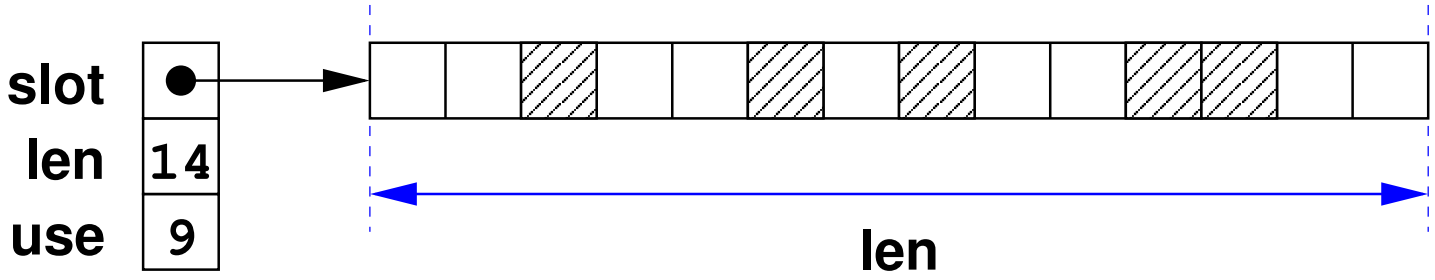
- 格納や削除は検索と整列が占めている

	格納	削除
整列配列	(拡張+)整列	検索+整列
非整列配列	(拡張)	検索
リスト	(malloc)	検索

  - ◇ 整列配列は整列の処理時間が長い
  - ◇ 非整列配列やリストは検索の処理時間が長い
- 整列の速い整列配列、あるいは検索の速い非整列配列かリスト、が必要

ハッシュの導入 *(cont.)*

- 今回は配列＋ハッシュ関数 (オープンハッシュ)
  - 『検索の速い非整列配列』
  - 格納場所の割り当てをハッシュ関数で行う
    - 全探索より速い
  - 整列しない
  - 領域確保も行う



ハッシュ関数

- キーから数字を生成する

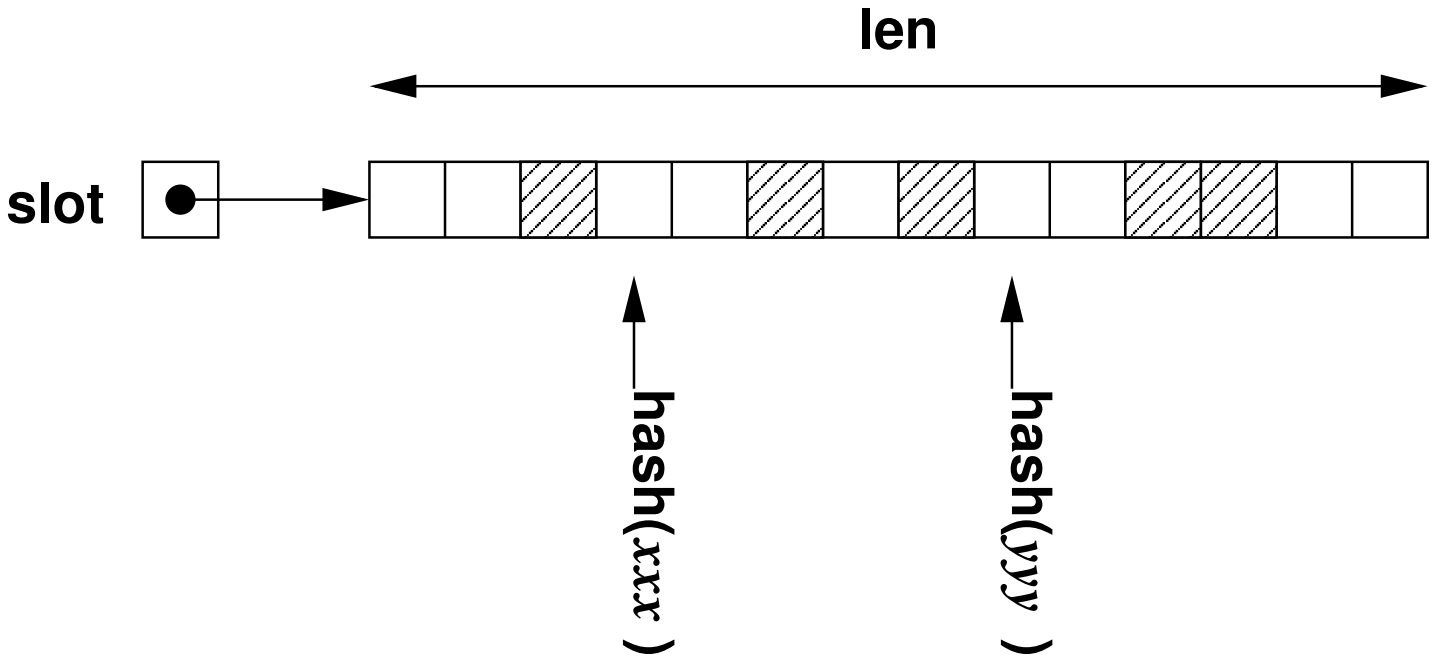
```
H(key)
```

- 原則として一意に定まればなんでもよい
- (典型的な)補助的要望
  - キーが短くても値が塊まらない
  - 似たようなキーを与えても違う値になる
- 今回のキーは文字列、文字を使って数字を作る
- 配列に納めるため、最後に剰余をとる

ハッシュ関数 *(cont.)*

```
int hash(char *src, int radix) {
    unsigned char *p;  int c=1;
    unsigned int sum=1027;
    p = (unsigned char *)src;
    while(*p) {
        sum <<= 1;
        sum += *p * c;
        c += 2;  p++;
    }
    return sum%radix;  }
```

格納



ハッシュ関数の戻り値を起点に格納場所を探す

- 使用中(キーが空ではない)なら隣に移動
- 端まできたら折り返す pos = (pos+1) % len

```
int idict_add(idict_a*dict,char*xkey,int xvalue) {
  int pos;
  if(dict->use>=dict->len) idict_expand(dict);
  pos = hash(xkey, dict->len);
  while( dict->slot[pos].key ||
    (dict->slot[pos].key &&
    !dict->slot[pos].key[0])) {
    pos = (pos+1)%dict->len;
  }
  if(!dict->slot[pos].key) dict->use++;
  else free(dict->slot[pos].key);
  dict->slot[pos].key = strdup(xkey);
  dict->slot[pos].value = xvalue;
  return 0;  }
```

初期化

配列の途中から使うこと、使用中かどうか判定する必要から、キーの初期化が重要

```
int idict_init(idict_a *dict) {
    int i;
    dict->len  = IDICTLEN;
    dict->slot = (idict_c*)malloc(
        sizeof(idict_c)*dict->len);
    dict->use  = 0;
    for(i=0;i<dict->len;i++)
        dict->slot[i].key = NULL;
    return 0; }
```

領域拡張

```
int idict_expand(idict_a *dict){
  idict_c *newslot, *oldslot;
  int i, newlen, oldlen;
  newlen = dict->len*2;
  newslot = (idict_c*)malloc(
                  sizeof(idict_c)*newlen);
  for(i=0;i<newlen;i++)
    newslot[i].key = NULL;
  oldslot = dict->slot; oldlen  = dict->len;
  dict->slot = newslot; dict->len  = newlen;
  dict->use  = 0;
```

領域拡張 *(cont.)*

```
  for(i=0;i<oldlen;i++) {
    if(oldslot[i].key && oldslot[i].key[0])
      idict_add(dict, oldslot[i].key,
        oldslot[i].value);
  }
  free(oldslot);

  return 0;
}
```

- 新しい領域を確保して、改めて add で格納し直す

検索

ハッシュ関数の戻り値を起点に格納場所を探す

- キーがあったら比較
- キーがない、あるいは該当しないかなら隣に移動
- 端まできたら折り返す pos = (pos+1) % len
- (念のため)一周したらあきらめる

検索 *(cont.)*

```
idict_c* idict_findpos(idict_a*dict,char *xkey) {
    int i=0, pos;  idict_c *p, *ppos=NULL:
    pos = hash(xkey, dict->len);
    while(i<dict->len+2) {
        p = &dict->slot[pos];
        if(p->key && strcmp(p->key, xkey)==0) {
            ppos = p;  break;
        }
        pos = (pos+1)%dict->len;
        i++;
    }
    return ppos; }
```

削除

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

- free でキーを開放しても良いが、free を呼ぶ回数を減らしたいので、空にするだけ

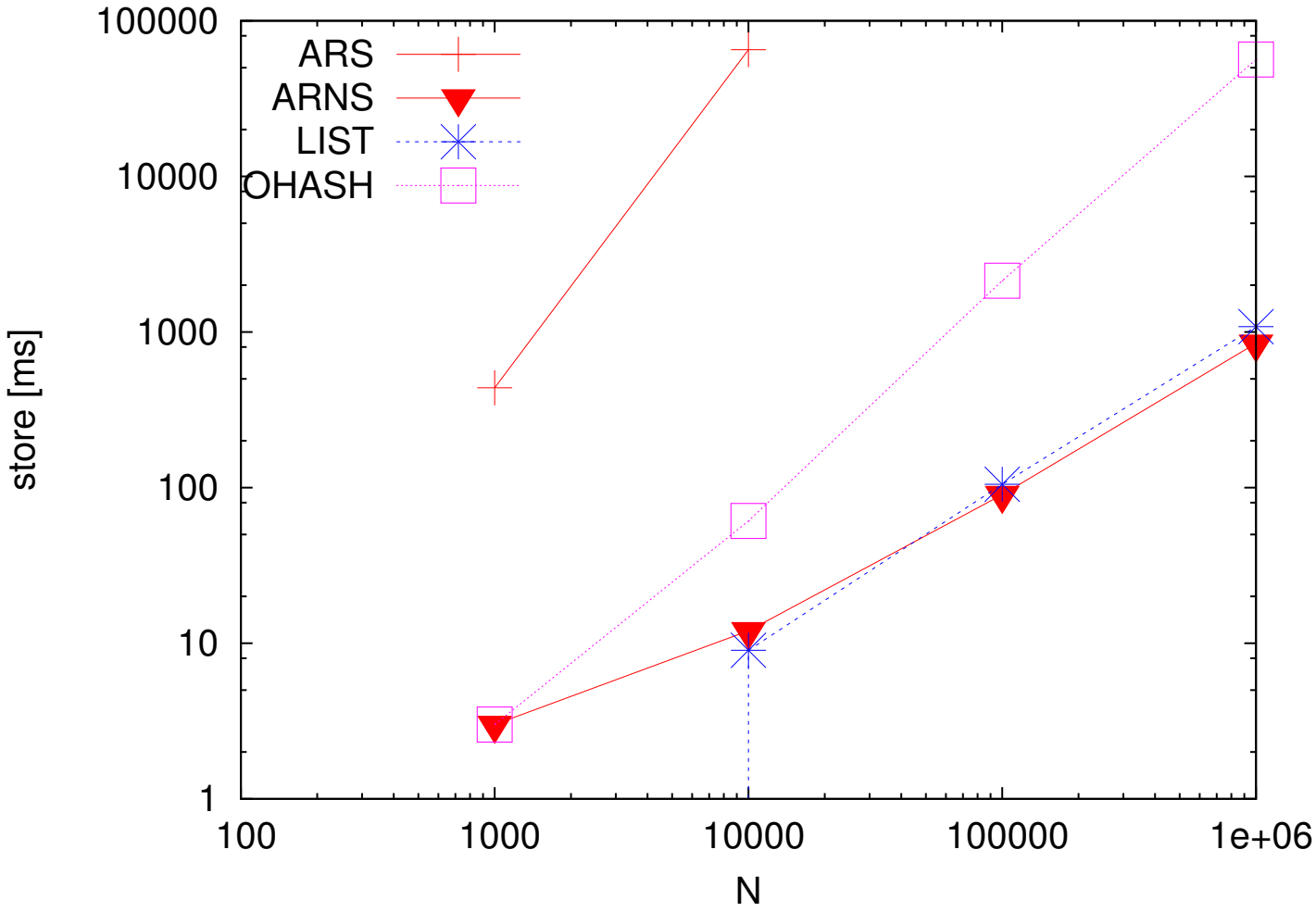
予想

	格納		削除	
整列配列	(拡張+)整列	極長	検索+整列	極長
非整列配列	(拡張)	短	検索	長
リスト	(malloc)	短	検索	長
ハッシュ配列	(拡張+)探索	短	検索	短

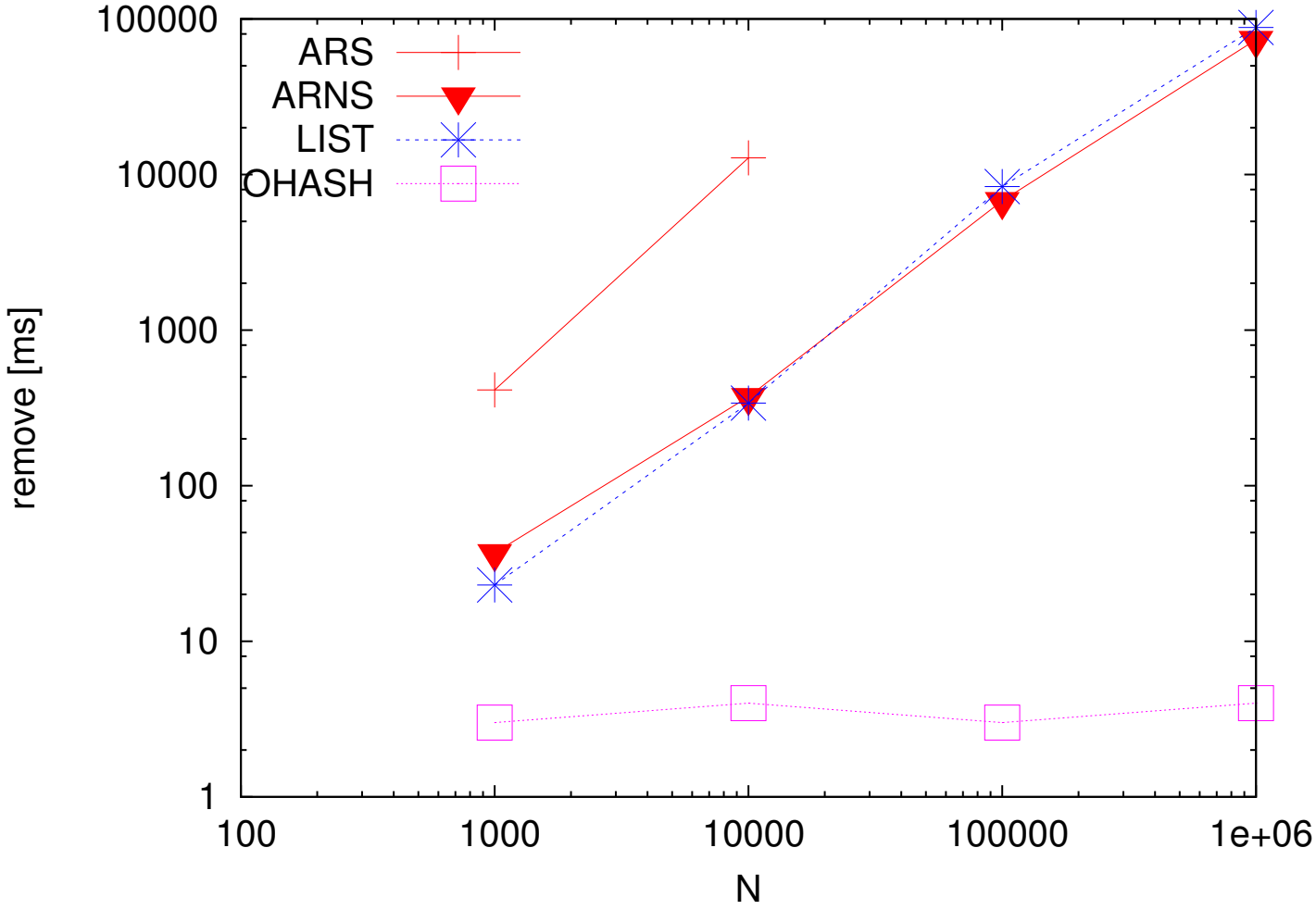
- 整列配列は整列の処理時間が長い
- 非整列配列やリストは検索の処理時間が長い
- ハッシュ配列の検索処理時間は短い

結果比較

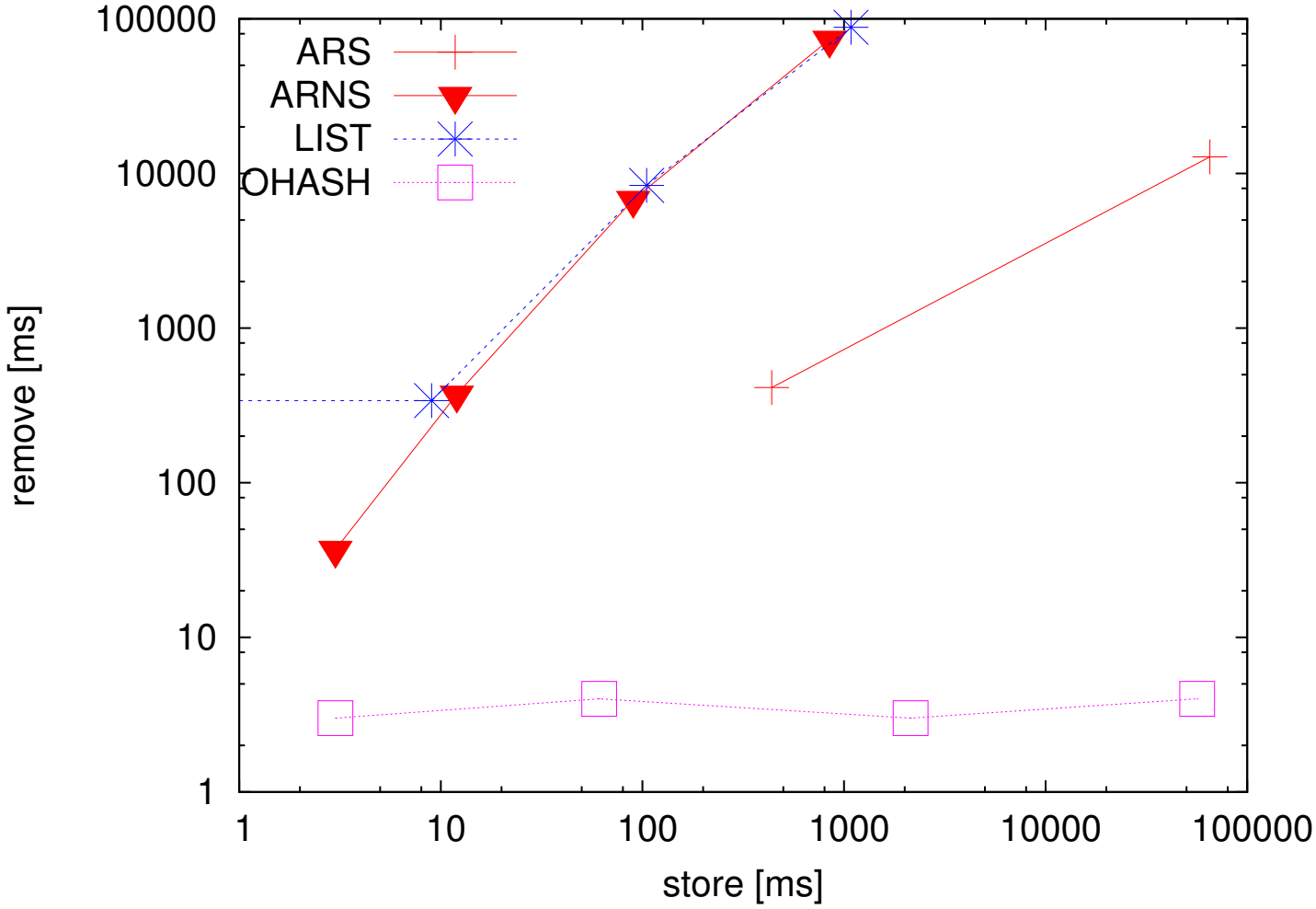
N	name	store	lookup	sort	lookup	remove
		N	*1k*	*N*	*1k*	*1k*
1000	ars	438	1	0	1	413
	arns	3	27	0	0	37
	list	0	26	–	–	23
	ohash	3	1	–	–	3
10000	ars	65213	2	11	2	12804
	arns	12	344	14	2	373
	list	9	397	–	–	340
	ohash	61	3	–	–	4



- ハッシュは非整列配列やリストに比べ遅い



- ハッシュは非整列配列やリストに比べ速く、安定している



- 削除(や検索)を優先するならハッシュ、格納なら非整列配列やリスト

結果比較 *(cont.)*

N	name	store *N*	lookup *1k*	lookup *1k*	remove *1k*
1000	arns	3	27	0	37
	list	0	26	–	23
	ohash	3	1	–	3
10000	arns	12	344	2	373
	list	9	397	–	340
	ohash	61	3	–	4
100000	arns	90	6800	–	6779
	list	105	7891	–	8376
	ohash	2139	4	–	3
1000000	arns	847	74546	6	73633
	list	1082	84781	–	88249
	ohash	56442	4	–	4

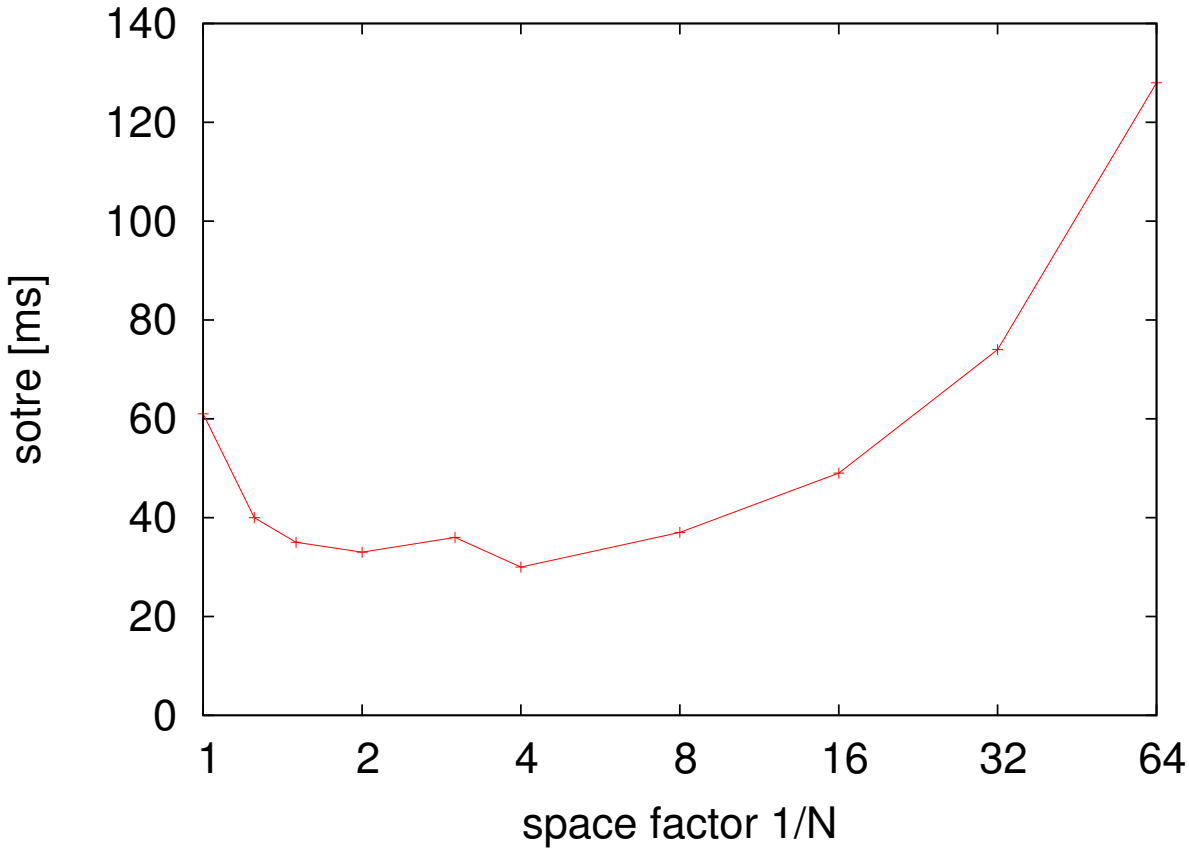
領域獲得因子

- 領域を大きくとるとハッシュ値の衝突が減る
- idic_add で領域を速めに拡大するコードに変更

たとえば、3倍（使用率 1/3 を越えたら拡大する）

```
    if(dict->use>=dict->len/3) {
        idict_expand(dict);
    }
```

- ある程度効果がみられるが...



- 1倍より大きければ効果が現れる
- 大きすぎると1倍より悪くなる

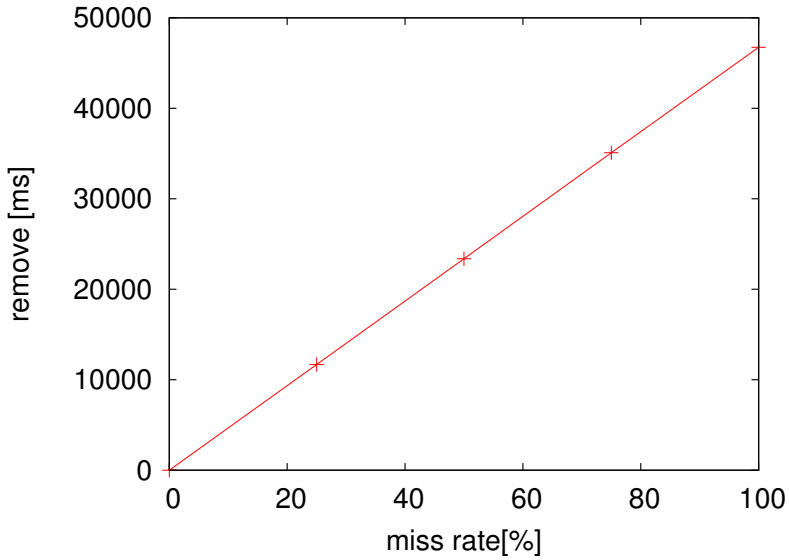
領域獲得因子 *(cont.)*

N	name	store	lookup	sort	lookup	remove
		N	*1k*	*N*	*1k*	*1k*
10000	ohash x1	61	3	–	–	4
	ohash x2	33	1	–	–	1
	ohash x3	36	1	–	–	2
	ohash x4	30	3	–	–	2
	ohash x8	37	3	–	–	2
	ohash x16	49	3	–	–	2
	ohash x32	74	3	–	–	2
	ohash x64	128	3	–	–	2

ミス率 対 処理時間

この実装はキーが見付からない（ミス）場合は全探索
N=10万、1000個削除の際のミス率と削除処理時間

率%	数	時間[ms]
0	0	1
25	250	11685
50	500	23373
75	750	35094
100	1000	46743



- 処理時間がミス率に比例している

補足

- ハッシュで求めた場所が使用中(衝突)の処理
  - ◇ 隣を探す（今回の実装）
  - ◇ 間隔をあけて探す
  - ◇ 別のハッシュ関数で再割り当て
    ※ 探索の終了条件に注意すること
- 『検索の速いリスト』の方向に進む場合は、スキップリスト等が利用できる
- 『整列の速い整列配列』は前述のように、qsortではなく追加時に挿入場所を探索することもできる

演習

1) 先のプログラムを実際に作成せよ★★
   - 最低限 new, init, expand, findpos, add, del が必要
2) ハッシュ関数をいくつか作成して、性能への影響を調べよ★★
3) ミス時の処理時間を減らすため、全探索せずにすむプログラムを作れ★★
4) スキップリストで似たような辞書プログラムを作れ★★★