# ダブル配列による自然言語辞書の高速更新法

大野将樹　　森田和宏　　泓田正雄　　青江順一
徳島大学工学部知能情報工学科
{oono,kam,fuketa,aoe}@is.tokushima-u.ac.jp

## １．はじめに

キー検索とは，キーを見出しとしてその関連情報であるレコードを探す技法であり，データベースシステムや自然言語処理システムなど非常に多くの分野で利用され，計算機による情報処理の基礎となるものである．キー検索法にはハッシュ法，Ｂ木法などその用途に応じて様々なものがあるが，なかでもトライ法[1]は，キーの文字単位に検索できること，共通接頭辞が併合されているという特徴から，スペルチェッカ，形態素解析辞書，かな漢字変換辞書など，大規模な自然言語辞書として利用されている．

このトライを効率的に実現するデータ構造にダブル配列[2]がある．ダブル配列は，トライ節の遷移を定数時間 O(1)で探索することができ，極めて高速なキー検索を実現できる．しかし，キーの更新速度は実用的といえず，特に，ダブル配列中に未使用領域が増加すると著しく更新速度が低下するので，その応用分野が限定されてきた[3].

本論文では，ダブル配列を動的キー検索法として確立するため，未使用領域が増加した状態でも更新速度が低下しない高速キー更新法を提案する．

## ２．ダブル配列とその問題点

### ２．１　トライとダブル配列

トライ法は木構造によりキー集合を表現するキー検索法である．キー集合 K={back#, bad#, badge#, bg#}に対するトライを図１に示す. キー末端の記号#はトライの葉とキーを一対一に対応させるためのものである．トライによるキー検索は１文字ごとに行われ，トライの節（node）の遷移を計算量 O(1)で確認できるならば，最悪の検索時間はキーの文字数に

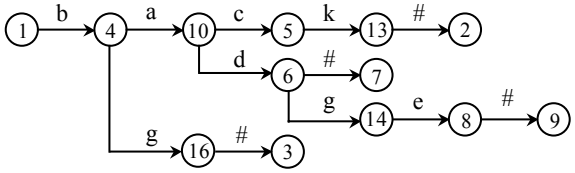


**図１**　キー集合 K に対するトライ

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BASE | 1 | -1 | -4 | 8 | 1 | 6 | -2 | 8 | -3 | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| CHECK | 1 | 13 | 16 | 1 | 10 | 10 | 6 | 14 | 8 | 4 | -12 | -15 | 5 | 6 | -17 | 4 |

**図２**　キー集合 K に対するダブル配列

依存し，高速な検索法となる．ダブル配列は２つの配列 BASE，CHECK を用いて，トライの節 s（s は節番号）から文字 a による節 t への遷移（g(s,a)=t と表す）を次式で定義する．

t=BASE[s]+code(a)，　CHECK[t]=s　　(1)

すなわち，節 s の遷移先 t は，BASE[s]と遷移文字 a の内部表現値 code(a)の和によって計算され，節 t が節 s からの遷移であることを CHECK[t]に s を格納することによって定義する．ダブル配列による g(s,a)=t の確認は式(1)より計算量 O(1)となるので高速なトライ検索が実現できる．図１のトライをダブル配列で表現したものを図２に示す．トライの節 n が葉ならば BASE[n]を負数とすることにより内部節と区別する．ダブル配列の最大の要素番号を MAX，未使用要素番号を昇順に $r_1, r_2,...,r_m$ とするとき，全ての未使用要素は次のリストにより連結される．

$CHECK[r_i] = -r_{i+1}$　$(1 \leqq i \leqq m-1)$

$CHECK[r_m] = -(MAX+1)$

$r_1$ はリストの先頭を示す LIST_HEAD に保持される．この未使用要素リストは，キーの追加，削除時に使用される．

## ２．２　検索アルゴリズム

ダブル配列からキーX=$a_1$ $a_2$...$a_n$ $a_{n+1}$，$a_{n+1}$=#を検索するアルゴリズム SEARCH(X)を以下に示す.

**[ 関数 SEARCH(X) ]**

**手順（S-1）:**｛初期化｝

ダブル配列の現在処理中のインデックス番号を示す変数 index を１に，入力キーの文字位置を示す変数 pos を１に初期化する.

**手順（S-2）:**｛遷移の確認｝

next=BASE[index]+code($a_{pos}$);

next>MAX または CHECK[next] ≠ index ならば g(index, $a_{pos}$)≠next なので FALSE を返し終了する.

**手順（S-3）:**｛終了判定｝

BASE[next] ≧ 0 ならば次の遷移を確認するため index=next，pos=pos+1 とし手順（S-2）へ.

BASE[next]<0 ならば，キーX の検索が成功するので TRUE を返し終了する.　　（関数終）

## ２．３　追加アルゴリズム

ダブル配列へキーX=$a_1$ $a_2$...$a_n$ $a_{n+1}$，$a_{n+1}$=#を追加するアルゴリズム INSERT(X)を以下に示す.

**[ 関数 INSERT(X) ]**

**手順（I-1）:**｛キーの登録確認｝

SEARCH(X)によりキーX を検索し，登録済みならば終了．未登録ならば検索失敗時の遷移を g(index, $a_{pos}$)=next として手順（I-2）へ.

**手順（I-2）:**｛衝突の回避｝

next が未使用要素ならば手順（I-3）へ．next が使用要素ならば，衝突を起こしている全ての遷移 g(index,c)=oldpos（next∈oldpos）を未使用要素へ移動する．要素 oldpos は ADD_LINK(oldpos)により未使用要素とする．移動先となる全ての未使用要素 newpos は DEL_LIST(newpos)により使用要素とする.

**手順（I-3）:**｛節の追加｝

遷移 g(index,$a_{pos}$...$a_{n+1}$)=remain をダブル配列に追加する．DEL_LIST(remain)により未使用要素を使用要素とし終了する.　　（関数終）

**[ 関数 ADD_LIST(index) ]**

**手順（A-1）:**｛未使用要素リストの先頭へ挿入｝

index<LIST_HEAD の場合，index が未使用要素リストの先頭になるので，BASE[index] に 0，CHECK[index]に-LIST_HEAD を格納し, LIST_HEAD を index に変更して終了する.

index≧LIST_HEAD の場合，手順（A-2）へ.

**手順（A-2）:**｛直前の未使用要素を探索｝

LIST_HEAD から未使用要素リストをたどり，prev<index<-CHECK[prev]なる prev を探索する.

**手順（A-3）:**｛未使用要素リストへ追加｝

BASE[index]=0，CHECK[index]=CHECK[prev]とし，CHECK[prev]を－index に変更する.　　（関数終）

**[ DEL_LIST(index) ]**

**手順（E-1）:**｛未使用要素リストの先頭を除外｝

index=LIST_HEAD の場合, 未使用要素リストの先頭が使用されるので，LIST_HEAD を-CHECK[index]に変更し終了する.

index≠LIST_HEAD の場合，手順（E-2）へ

**手順（E-2）:**｛直前の未使用要素を探索｝

LIST_HEAD から未使用要素リストをたどり，-CHECK[prev]=index なる prev を探索する.

**手順（E-3）:**｛未使用要素リストから除外｝

CHECK[prev]を CHECK[index]に変更し，index を未使用要素リストから除外する.　　（関数終）

## ２．４　削除アルゴリズム

ダブル配列からキーX=$a_1$ $a_2$...$a_n$ $a_{n+1}$，$a_{n+1}$=#を削除するアルゴリズム DELETE(X)を以下に示す.

**[ 関数 DELETE(X) ]**

**手順（D-1）:**｛キーの登録確認｝

SEARCH(X)によりキーX を検索し，未登録ならば終了する．登録済みならば手順（D-2）へ.

**手順（D-2）:**｛節の削除｝

削除キーX のみを定義しているすべての要素を ADD_LINK(index)で未使用要素とする.　（関数終）

## ２．５　ダブル配列の問題点

ダブル配列の問題点は，ADD_LIST と DEL_LIST の処理時間である．要素 index を未使用要素リストに追加する ADD_LIST(index)は，手順（A-2）において要素 index の直前の未使用要素 prev を発見するために，未使用要素リストを先頭から走査しており，最悪の場合，未使用要素の数に依存した計算量となる．また，未使用要素 index を未使用要素リストから除外する DEL_LIST(index)も，手順（E-2）において未使用要素 index の直前の未使用要素 prev を発見するために，未使用要素リストを先頭から走査するので，未使用要素数に依存した計算量となる．

## ３．双方向未使用要素リンクによるダブル配列の高速更新アルゴリズム

ADD_LIST および DEL_LIST を高速にするため，発生順かつ双方向の未使用要素リストを定義する．

［**定義**］未使用要素がインデックス番号 $r_1, r_2, ..., r_m$ の順で作成されたとき，全ての未使用要素は次のリストにより連結される．

$BASE[r_1] = -(MAX+1)$

$BASE[r_i] = -r_{i-1}$　　$(2 \leqq i \leqq m)$

$CHECK[r_i] = -r_{i+1}$　$(1 \leqq i \leqq m-1)$

$CHECK[r_m] = -(MAX+1)$

未使用要素リストの先頭 $r_1$，$r_m$ は大域変数 LIST_HEAD, LIST_TAIL が保持する．　（定義終）

未使用要素リストを双方向に連結することにより DEL_LIST(index)における未使用要素 index の直前の未使用要素 prev の探索は，prev=－BASE[index]により容易に確認できるので，O(1)の計算量となる．また，未使用要素リストを未使用要素の発生順に連結することにより，ADD_LIST(index)における使用要素 index の直前の未使用要素 prev の探索は，prev=－LIST_TAIL により容易に確認できるので，O(1)の計算量となる．

未使用要素リストを更新する新しい関数 ADD_LIST，DEL_LIST を以下に示す．

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BASE | 1 | -1 | -4 | 8 | 1 | 6 | -2 | 8 | -3 | 1 | -17 | -15 | 1 | 2 | -11 | 2 |
| CHECK | 1 | 13 | 16 | 1 | 10 | 10 | 6 | 14 | 8 | 4 | -15 | -17 | 5 | 6 | -12 | 4 |

**図３**　作成順かつ双方向にリンクされた未使用要素リストの例

［**関数 ADD_LIST(index)**］

**手順（P-1）:**｛直前の未使用要素を発見｝

index の直前の未使用要素番号 LIST_TAIL を prev にセットする．

**手順（P-2）:**｛未使用要素リストへ追加｝

BASE[index]に－prev, CHECK[index]に CHECK[prev]をセットし，CHECK[prev]を－index に変更し終了．

（関数終）

［**関数 DEL_LIST(index)**］

**手順（Q-1）:**｛未使用要素リストの先頭を除外｝

index の直後の未使用要素番号－CHECK[index]を next にセットする．

index=LIST_HEAD の場合, 未使用要素リストの先頭が使用されるので，LIST_HEAD を next に，BASE[next]を BASE[index]に変更し終了する．

index≠LIST_HEAD の場合，手順（Q-2）へ．

**手順（Q-2）:**｛直前の未使用要素を発見｝

index の直前の未使用要素番号 LIST_TAIL を prev にセットする．

**手順（Q-3）:**｛未使用要素リストの末尾を除外｝

index=LIST_TAIL の場合，未使用要素リストの末尾が使用されるので，LIST_TAIL を prev に変更し，CHECK[prev]に CHECK[index]をセットし終了．

index≠LIST_TAIL}の場合，手順（Q-4）へ．

**手順（Q-4）:**｛未使用要素リストから除外｝

CHECK[prev] を CHECK[index] に，BASE[next] を BASE[index]に変更し終了する．　（関数終）

［**例**］図３のダブル配列において，ADD_LIST(9)が呼び出された場合の例を示す．まず，手順（P-1）で，prev=LIST_TAIL=12 とし，手順（P-2）で，BASE[9]=－12，CHECK[9]=CHECK[12]=－17，CHECK[12]=

−9 となり，要素 9 が未使用要素リストの末尾に追加される．

また，DEL_LIST(15)が呼び出された場合の例を示す．まず，手順（Q-1）で，15 の直後の未使用要素番号−CHECK[15]=12 を next に，手順（Q-2）で 15 の直前の未使用要素番号−BASE[15]=11 を prev にセットする．手順（Q-4）で CHECK[prev]=CHECK[11] を CHECK[15]=−12 に，BASE[next]=BASE[12] を BASE[15]=−11 に変更し，要素 15 が未使用要素リストから除外される．（例終）

## ４．評価

### ４．１　理論的評価

ダブル配列の未使用要素の数を m とする．ダブル配列への追加の最大時間計算量は，衝突した節の移動先を探索する時間と未使用要素リンクの更新時間（関数 ADD_LINK，DEL_LINK の計算量）に依存する．従来法は衝突回避の探索に O(m)，未使用要素リンクの更新に O(m)かかるので $O(m^2)$となる．提案法は未使用要素リンクの更新が O(1)ですむので O(m)となる．ダブル配列の削除の最大時間計算量も未使用要素リンクの更新時間に依存する．よって，従来法は O(m)，提案法は O(1)となる．

### ４．２　実験による評価

提案法の構成システムは約 500 行の C 言語で記述されており，OS:Windows2000，CPU:PentiumⅢ1GHz 上で稼働している．実験に用いたデータは，EDR 電子化辞書の日本語単語辞書である．

表１に単語１万語を徐々に追加していった場合の追加時間と記憶量を示す．追加時間はほぼ変化ないが，これは追加過程では未使用要素が極めて少ないため，処理速度に大きな差がでなかったためである．また提案法は記憶量が若干増加するが，これは未使用要素リンクが作成順となったことが要因である．

表２に単語１万語をダブル配列に登録後，徐々に削除していった場合の実験結果を示す．表２の結果から，提案法は従来法より約 3〜60 倍高速に削除できていることがわかる．これは提案法により，未使用要素リストの更新時間が未使用要素の総数 m に依存しなくなったことに起因する．

**表１**　追加に対する実験結果

| 追加キー数 | 1,000 | 3,000 | 5,000 | 7,000 | 9,000 |
|---|---|---|---|---|---|
| 追加時間[秒] | | | | | |
| 従来法 | 0.1 | 0.3 | 0.8 | 1.1 | 1.4 |
| 提案法 | 0.1 | 0.3 | 0.9 | 1.2 | 1.5 |
| 記憶量[MB] | | | | | |
| 従来法 | 0.5 | 1.4 | 2.2 | 3.9 | 4.9 |
| 提案法 | 0.5 | 1.6 | 2.5 | 4.1 | 5.1 |

**表２**　削除に対する実験結果

| 削除キー数 | 1,000 | 3,000 | 5,000 | 7,000 | 9,000 |
|---|---|---|---|---|---|
| 削除時間［秒］ | | | | | |
| 従来法 | 0.4 | 3.6 | 16.6 | 51.7 | 91.2 |
| 提案法 | 0.1 | 0.4 | 0.8 | 1.1 | 1.5 |

提案法は従来法と同等の記憶量を維持しつつ，極めて高速な削除を実現しており，ダブル配列の更新法として有効であるといえる．

## ５．おわりに

本論文では，未使用要素リストを未使用要素作成された順かつ双方向に連結することにより，大量の未使用要素が生成された場合でも高速にキー更新を行う手法を提案した．今後の課題は，提案法を実用システムに組み込み，有効性を確認することである．

## 参考文献


[1]Fredkin, E.：Trie Memory, Comm.ACM, Vol.3, No.9, pp.490-500 (1960).

[2]青江順一：ダブル配列による高速ディジタル検索アルゴリズム，電子情報通信学会論文誌 D, No.9, pp.1592-1600 (1998).

[3]森田和宏，泓田正雄，大野将樹，青江順一：ダブル配列における動的更新の効率化アルゴリズム，情処学会論文誌, Vol.42, No.9, pp.2229-2238 (2001).