Vol.2016-DBS-163 No.12
Vol.2016-IFAT-123 No.12
2016/9/14

動的ダブル配列辞書における実用的な再構成法

神田 峻介[1,a)]　藤田 勇磨[1]　森田 和宏[1]　泓田 正雄[1]

概要：ダブル配列とは，文字列をキーとする辞書を実現するために広く用いられるデータ構造である．動的辞書として利用する場合，キーの削除に伴う記憶効率の低下が問題とされてきたが，ダブル配列の再構成により，この問題が解決されることが先行研究により示されている．本稿では，従来の再構成法に関して，その実行時間が実用において問題となる点について述べ，それを改善するための新しい再構成法を提案する．1,000 万件以上のキーから構成される辞書を用いた実験により，従来手法では 2 分以上要した再構成を，提案手法では 1 秒未満でおこなえることを示す．加えて，提案手法による再構成は検索速度の向上にも繋がることを示す．

Practical Rearrangement of Dynamic Double-array Dictionaries

Shunsuke Kanda[1,a)]　Yuma Fujita[1]　Kazuhiro Morita[1]　Masao Fuketa[1]

Abstract: The double-array is a data structure widely used to implement dictionaries with string keys. While a previous dynamic double-array dictionary had a problem that its memory efficiency decreases with key deletion, earlier studies have solved it by rearranging the double-array. This paper describes that conventional rearrangement time becomes problematic in practice, and proposes new rearrangement methods. From experiments using a dictionary with over ten million keys, we show that our methods use less than one second for rearrangement while the conventional ones use over two minutes. In addition, we show that our rearrangement can improve search speed.

1. はじめに

トライ（Trie）[1] とは枝に文字を付随した順序木であり，文字列集合を管理するために広く用いられる．文字列の接頭辞を併合することで構築され，根から葉を繋ぐ枝上の文字を連結することにより登録文字列が復元される．また，各文字列に一意に対応する節点が存在するため，文字列から固有の識別子への写像を与えることができる．単純な完全一致検索に加え，自然言語処理などで必要とされる接頭辞ベースの検索も提供できることから，文字列をキーとした辞書としての利用が多く見られる．

このトライを用いた辞書を実現するために，ダブル配列（Double-array）[2] は現在も広く用いられている．ダブル配列とは，極めて高速な検索を利点とするデータ構造であり，静的辞書と動的辞書の両方の用途に対して多くの研究がなされてきた．静的辞書については，主に記憶効率と検索速度の向上を目的とした研究が多く見られる [3, 4]．動的辞書と比べ実装も単純であることから，頻繁なキーの更新（追加や削除）を必要としない語彙辞書などで利用されている [5,6]．動的辞書については，主に更新速度の改善を目的とした研究が多く見られる [7–10]．当初のダブル配列は，更新速度が低速かつ不安定という問題を抱えており，動的辞書としての利用はあまり報告されていなかった．しかし，矢田らの研究 [9] において，静的辞書と同等の構築時間でキーの逐次追加が可能であることが示されるなど，更新に潜むボトルネックの多くが解決され，近年では動的辞書としての利用がいくつか報告されている [11, 12]．

動的辞書に関して，更新速度の改善を目的とした研究の他に見られるのが，記憶効率改善のための再構成法に関する研究である．ダブル配列は，ハッシュ表のように未使用の要素を含むことが前提にあり，その充填率（使用されている要素の割合）が記憶効率を左右するデータ構造である．

[1] 徳島大学大学院先端技術科学教育部
[a)] shnsk.knd@gmail.com

Vol.2016-DBS-163 No.12
Vol.2016-IFAT-123 No.12
2016/9/14

トライにおける節点の削除は，その節点に対応する要素を未使用にすることで実現されるため，キー削除の繰り返しに伴う充填率の低下が問題となる．この問題に対し，未使用要素を詰め直すことによりダブル配列を再構成し，充填率を改善するという措置が考えられてきた [13–15]．矢田らによる再構成法 [15] を用いれば，充填率を 100%近くに維持できることが実験により示されている．

これまでの再構成法は，常に充填率を高く維持するという目的において，キーを削除するたびに再構成をおこなう手順の上で評価を得てきた．しかし，再構成のために実際に必要な操作は多く，単純にキーを削除する場合と比べると多くの時間を必要とする．また，削除によって発生した未使用要素は，その後の追加において再利用の対象となるため，削除時間を犠牲にしてまで常に充填率を 100%に保つ利点はあまりない．そのため実用上は，充填率に対して閾値を設定しておき，それをトリガーとして再構成を実行するといった利用が考えられる．

本稿では，ダブル配列を用いた動的辞書の更なる実用化を目的とし，実用において効率的な再構成法を提案する．提案手法に対する評価は実験により与え，実行時間が大幅に短縮されることを示す．加えて，提案手法による再構成が，その後の検索速度の向上にも繋がるということを示す．

2. 準備

本節では，本研究に関連する従来研究をいくつか紹介する．はじめに，基本的な定義を以下に与える．本稿では，トライの節点数を n で表す．枝に付随する文字はアルファベット $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ 上で表される．配列の要素は 0 から始まる添字で指定し，対数の底は 2 で統一する．

2.1 ダブル配列

ダブル配列は，BASE，CHECK と呼ばれる 2 つの配列を用いて，トライを表現するデータ構造である．$\mathrm{BASE}[s]$ と $\mathrm{CHECK}[s]$ は節点 s に対応しており，節点 s に対応する要素を要素 s と表す．任意の内部節点 s から出ている文字 c の枝が子 t を指す状態に対し，次式を満足する．

$$\mathrm{BASE}[s] \oplus c = t,\ \mathrm{CHECK}[t] = s \qquad (1)$$

図 1 では，文字 c_1，c_2 により節点 s から子 t_1，t_2 をそれぞれに指す状態を，ダブル配列により表した例を示している．節点 s から文字 c_1 により遷移される子 t_1 を特定する場合，$t_1 \leftarrow \mathrm{BASE}[s] \oplus c_1$ により候補となる子が算出でき，$\mathrm{CHECK}[t_1]$ と親 s の値を比較することで，その子が存在するかを識別できる．逆に親の特定は，CHECK 値を参照するだけでよく，極めて少ない演算回数で節点間の移動ができることがダブル配列の最大の利点である．

一方で，ダブル配列は BASE 値を基に，式 (1) を満たすように節点を配置し構築されるため，未使用の要素を含むことが前提にある．ここで未使用要素について，いくつか定義を与えておく．要素 s が使用されているか否かを，配列 $\mathrm{USED}[s] \in \{\mathrm{TRUE}, \mathrm{FALSE}\}$ を用いて表す．未使用要素とは配列途中に含まれるものであり，ダブル配列の配列長を N とすると $\mathrm{USED}[N-1] = \mathrm{TRUE}$ である．また，未使用要素の添字集合を $R = \{0 \le s < N \mid \mathrm{USED}[s] = \mathrm{FALSE}\}$ と表す．未使用要素数 $|R| = m$ とすると，$N = n + m$ であり，ダブル配列の充填率は n/N で表される．この充填率が高いほど記憶効率の良いダブル配列といえる．

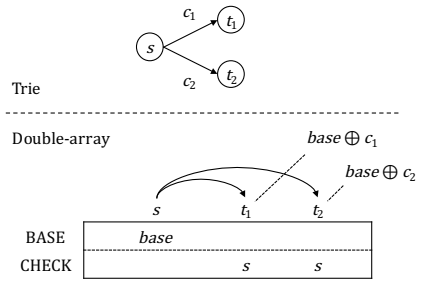


図 1 トライとそれを表したダブル配列．

辞書としてダブル配列を用いる場合，圧縮と高速化を目的として MP トライ（Minimal Prefix Trie）[2] がよく表現される．MP トライでは，各キーを識別するために必要な最小の接頭辞集合のみを木構造として管理し，残りの接尾辞を文字列として別領域で管理する．接尾辞の参照は，各葉にリンクを与えることで実現され，ダブル配列では葉の BASE 値によってこのリンクを表現する．MP トライでは，単純なトライと比べ少ない節点数で辞書を実現できるため，コンパクトにダブル配列辞書を実現できる．また，接尾辞を文字列として照合できるため，検索も高速化される．

2.2 キーの削除と再構成法

ダブル配列では，削除キーのみと対応する節点における要素を未使用にすることで，キーの削除を実現する．この処理の問題点として，削除回数に比例して未使用要素数が増加し，充填率が低下することがあげられるが，この問題は矢田らによって提案された再構成法を用いることで解決される [15]．矢田らによる再構成手続き REARRANGE を Algorithm 1 に示す．また，Algorithm 1 で用いられる関数及び手続きを以下のように定義する．

- $\mathrm{EDGES}(s)$: 節点 s から出る枝の付随文字の集合を返す．
- $\mathrm{SHELTER}(s, base, E)$: 集合 E に含まれる全ての文字 c に対して，$\mathrm{USED}[base \oplus c] = \mathrm{TRUE}$ となる要素とその兄弟の要素を，$R - \{base \oplus c \mid c \in E\}$ 中の未使用要素に移動するか，もしくは配列の末尾より後方に移動する．結果として，要素 s を移動させた場合は，その移動先を返し，それ以外はそのまま s を返す．
- $\mathrm{MOVE}(s, base, E)$: 集合 E に含まれる全ての文字 c に対して，要素 $\mathrm{BASE}[s] \oplus c$ を要素 $base \oplus c$ に移動する．

Algorithm 1 矢田らによる再構成手続き.

```
 1: procedure REARRANGE
 2:     while R ≠ ∅ do
 3:         s ← CHECK[N − 1]                          ▷ 末尾の親を特定
 4:         E ← EDGES(s)
 5:         base ← XCHECK(E)                          ▷ 圧縮要素の移動先を探索
 6:         if base = NIL then
 7:             break
 8:         s ← SHELTER(s, base, E)                   ▷ 衝突を回避
 9:         MOVE(s, base, E)                          ▷ 詰め直し
10: function XCHECK(E)
11:     for all r ∈ R do
12:         base ← r ⊕ E[0]                           ▷ BASE 値を逆算
13:         if ISTARGET(base, E) = TRUE then
14:             return base
15:     return NIL                                    ▷ 探索失敗
16: function ISTARGET(base, E)
17:     for all c ∈ E do
18:         s ← base ⊕ c
19:         if s = Root or N ≤ s then                 ▷ Root : 根 ID
20:             return FALSE
21:         if USED[s] = TRUE then
22:             if |E| ≤ |EDGES(CHECK[s])| then
23:                 return FALSE
24:     return TRUE
```

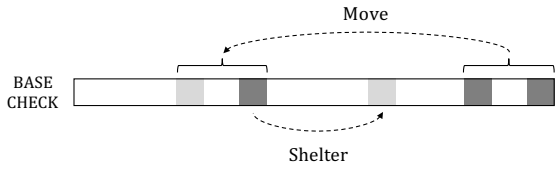


図 2 REARRANGE の挙動.

REARRANGE は，末尾の要素を前方に移動し，未使用要素を詰め直すことで充填率の改善を図る．このとき，式 (1) を満たすために，末尾の要素と一緒に兄弟の要素も移動する必要があるため，それらをまとめて圧縮要素と呼ぶことにする．REARRANGE では，関数 XCHECK を用いることで，以下のように圧縮要素の移動先を求めている．

XCHECK は，圧縮要素の移動先を探索する関数であり，結果として移動先を示す BASE 値を返す．R を走査し，未使用要素からの逆算により得られた BASE 値が，圧縮要素の移動先を示す値として適しているかどうかを，ISTARGET を用いて判断する．ISTARGET により圧縮要素の移動先候補として識別される要素は，未使用要素，もしくは圧縮要素数よりも兄弟の数が少ない要素である．

圧縮要素の移動は，XCHECK により決定された BASE 値を用いておこなわれるが，移動先に使用済み要素も含まれるため，前もって SHELTER により衝突を回避し，MOVE により詰め直しをおこなう（図 2）．SHELTER により移動される要素は，末尾の要素よりも兄弟が少ないため移動しやすく，この処理を繰り返すことで 100%近い充填率を達成できることが実験により示されている．

2.3 未使用要素探索の高速化

XCHECK における R の走査時間が，REARRANGE の実行時間を最も左右する．初期のダブル配列では，この走査のために配列全体を線形探索していたが，大規模な辞書に対してこの探索時間は致命的な問題となる．そのため，未使用要素を双方向循環連結リストとして構築しておき，未使用要素のみを走査できる状態で管理しておく Empty-Link（EL）法が一般的に用いられる [7,8].

仮に $R = \{r_1, r_2, \ldots, r_m\}$ としたとき，EL 法では BASE, CHECK の未使用要素を利用し，以下のように双方向循環連結リストを構築する．

$$\mathrm{BASE}[r_i] = \begin{cases} r_{i+1} & (1 \le i < m) \\ r_1 & (i = m) \end{cases}$$

$$\mathrm{CHECK}[r_i] = \begin{cases} r_{i-1} & (1 < i \le m) \\ r_m & (i = 1) \end{cases}$$

ここで，任意の r_i をリストの先頭として記憶しておくことで，$O(m)$ での走査が可能となる．

新たに未使用要素を加える場合は，リストの先頭要素と末尾要素の間に挿入し，削除する場合は，その未使用要素の前後を相互に連結するだけでよい．そのため，リストの更新は定数時間で実行できる．

2.4 枝列挙の高速化

REARRANGE に潜むボトルネックとして，枝列挙の実行時間もあげられる．単純な実装において EDGES(s) は，Σ に含まれる全ての文字 c に対して，節点 s から子が定義されているかを確認することにより，$O(\sigma)$ で実行されていた．実際には，キーをバイト文字列として扱うため，$\sigma = 256$ であり，この実行時間は極端に低速な訳ではない．しかし，REARRANGE のように何度も EDGES が呼び出される環境においては，重大なボトルネックとなり得る．

この問題は，矢田らによって提案された Node-Link（NL）法を用いることで解決できる [9]．NL 法では，新たに配列 CHILD と SIB を用いる．CHILD[s] は，節点 s とその最初の子とを繋ぐ枝上の文字を格納し，SIB[s] は，節点 s の次の兄弟とその親とを繋ぐ枝上の文字を格納する．このとき，節点 s の最初の子と次の兄弟は，BASE[s] $\oplus$ CHILD[s], BASE[CHECK[s]] $\oplus$ SIB[s] のように定数時間で求められ，$E \leftarrow$ EDGES(s) は，$O(|E|)$ で実行可能となる．ただし，追加で $2N \log \sigma$ ビット必要であり，NL 法の利用は，更新時間と記憶効率のどちらに重点を置くかによって決まる．

3. 実用的な再構成法

本節では，充填率が低下した状態における従来の再構成法の問題点を考察し，その解決策を提案する．また，違った手順による再構成法や高速化手法を提案する．

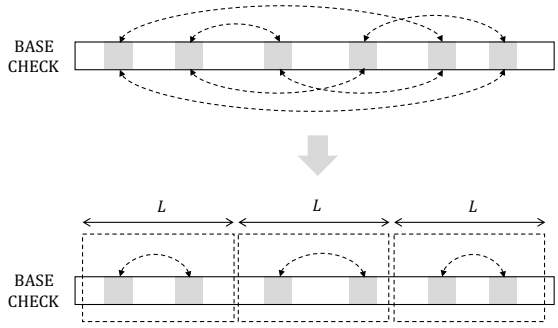


図 3 従来の EL 法と BL 法を適用した状態.

3.1 キャッシュ効率を考慮した EL 法

EL 法を用いることにより，XCheck における R の走査が $O(m)$ で実行可能となる一方で，新しい未使用要素はリストの末尾に挿入されるため，$r_1, r_2, \ldots, r_m$ と配列上の順序関係はなくなる．そのため，キーの削除が繰り返された後の連結リストは，$r_1, r_2, \ldots, r_m$ が配列上に隣接せず，結果として，XCheck でキャッシュミスが発生しやすい状態となる．

そこで，配列を長さ L のブロック毎に分割し，ブロック単位で連結リストを構築することにより，問題の解決を図る．具体的には，ブロック内の未使用要素で双方向循環連結リストを構築し，未使用要素を含む各ブロックについても，同じように双方向循環連結リストを構築する．結果として，同じブロックに含まれる未使用要素は連続して参照することができる（図 3）．本稿では，この手法を Block-Link（BL）法と呼ぶ．

BL 法では，以下のような理由で $L = 2^{\lceil \log \sigma \rceil}$ とすると効率が良い．XCheck における $base \leftarrow r \oplus E[0]$ について，$E[0] \in \Sigma$ より，r の下位 $\lceil \log \sigma \rceil$ ビットに対し排他的論理和した結果が $base$ である．すなわち，$L = 2^{\lceil \log \sigma \rceil}$ のとき，$\lfloor r/L \rfloor = \lfloor base/L \rfloor$ であり，r と $base$ は同じブロック内の値を表す．この $base$ を用いて，IsTarget では移動先候補を走査するが，文字 $c \in E$ に対する $s \leftarrow base \oplus c$ もまた，$base$ と同じブロック内を表す値である．結果として，ある r_i に対し IsTarget を呼び出すと，そのブロック内の要素は上位キャッシュに割り当てられる確率が高く，続く r_{i+1} 以降が連続して同じブロック内の要素であった場合，キャッシュ効率の向上が見込める．

この BL 法については，既に文献 [9] において同様の手法が提案されている．この文献では，キーの追加における更新時間の高速化および安定化を目的とし，各ブロックにおける未使用要素数や過去の探索失敗回数などを用いてブロックを分類することで，問い合わせに適した探索範囲の制御を実現している．ただし，このブロック分類を単純に Rearrange に応用することは難しく，本稿ではキャッシュ効率化による性能向上のみを評価する．

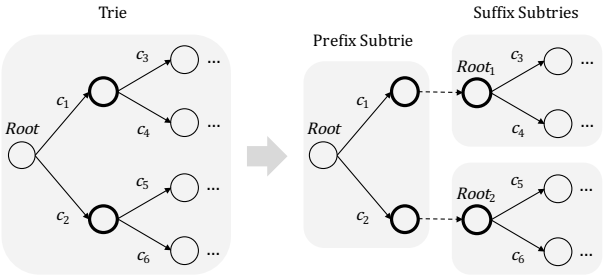


図 4 分解前後のトライ．太枠の節点は分解基準を表す.

3.2 静的な再構築による単純な再構成

従来研究では Rearrange のように，未使用要素を詰め直すことにより充填率の改善を図ってきた．一方で，移動先候補の探索や要素移動など，実際に必要とされる操作は多く，単純にダブル配列を静的に構築し直した方が，効率的であるとも考えられる[*1]．

静的な再構築における利点として，衝突回避のための要素移動が必要なく，結果的に親と子が隣接しやすいため，キャッシュ効率向上による検索の高速化が期待できることがあげられる．実際，動的にキーを逐次追加した場合に比べ，静的に構築した方が高速な検索をおこなえることが，文献 [9] において示されている．Rearrange を実行した場合，逐次追加以上の要素移動がおこなわれるため，静的再構築におけるキャッシュ効率化は，この文献の結果よりも顕著に現れることが予想される．

3.3 再構成の並列化

再構成を並列におこなうことで，実行時間をさらに短縮することができる．提案手法では，トライを単一の接頭辞部分トライ（Prefix Subtrie）と，複数の接尾辞部分トライ（Suffix Subtries）に分解して構築することにより，再構成の並列化を可能にする．図 4 は，トライを接頭辞部分トライと接尾辞部分トライに分解した例を示している．太枠の節点によりトライは分解されており，本稿ではこの節点を分解節点と呼ぶ．

接頭辞部分トライは，全てのキーに対し共通の根を持ち，分解節点と対応する各葉が接尾辞部分トライの根へのリンクを保持している．葉がリンクを保持するという機能が競合するため，MP トライではなく単純なトライとして構成される．接尾辞部分トライは，根が分解節点と対応し，分解節点以降のトライを従来通り MP トライとして構成する．検索は，接頭辞部分トライ，接尾辞部分トライの順に節点を移動することにより実現される．このように分解された各トライを，それぞれ個々に表現することにより，独立した複数のダブル配列を並列に再構成することができる．

トライの分解基準については，登録するキーの特徴に応じて予め設定すればよい．このとき，分解節点が余計に必

[*1] 静的構築の具体的な方法については，[16] を参照されたい.

要となる点を考慮し，接尾辞部分トライを大量に作らず，かつ並列化が有効に働くような分解基準を考える必要がある．自然言語処理における語彙集合など，特別に接頭辞に特徴を持たないキーを対象とする場合は，図 4 のように根から出る節点を分解節点とし，キーの 1 文字目を境に分解すればよい．実用において，接尾辞部分トライの数は高々 $\sigma = 256$ であり，頻出傾向によって各トライの節点数に違いはあるが，並列化が有効に働くことが予想される．

一方で，接頭辞に “`http://`” や “`https://`”，“`ftp://`” などが頻出する URL 集合を扱う場合は，1 文字目を境に分解すれば，極端に大きい接尾辞部分トライが構築されてしまう．このときは，予め接頭辞部分トライに頻出が予想される接頭辞を登録しておき，それによって定義された各節点から出る節点を分解節点とし，トライを分解すればよい．

4. 実験による評価

本節では，節 2 と節 3 で紹介した手法の様々な組み合わせに対し，再構成法としての評価を実験により与える．評価項目には再構成時間の他に，静的再構築によるキャッシュ効率化を評価するため再構成後の検索速度も加えた．

4.1 実験設定

実験に用いた計算機の構成は，Intel Core i7 4.0 GHz CPU，L2 cache 256 KB，L3 cache 8 MB であり，OS は OS X 10.11 である．実装言語は C++で，最適化オプションとして`-O3` を指定し，Apple LLVM version 7.3.0（clang-703.0.31）を用いてコンパイルした．実行時間の計測には，`std::chrono::duration_cast` を用いた．並列化の実装には，`std::thread` を用いた．起動できる最大スレッド数は 8 である．

辞書の構成には，2015 年 2 月時点での英語版 Wikipedia のタイトル記事集合を用いた[*2]．キー数が 11,519,354 件，キーの平均長が 20.7 バイトである．これらキーを無作為の順に逐次追加することによって辞書を構築し，そこから 10%ずつ無作為にキーを削除することによって得られた 10 パターンの各辞書に対して再構成法を適用した．並列化のためのトライ分解は，図 4 のようにキーの 1 文字目を境に分解にした．このとき，接尾辞部分トライの数は 107 となった．再構成が実行される前の各辞書の充填率の状態を表 1 に示す．トライの分解による充填率の変化はあまり見られなかったため，この表では分解していない状態の充填率のみを示している．

詰め直しによる再構成法 [15] と静的再構築による再構成法の 2 つの手法に対し，NL 法 [9]，BL 法，並列化をそれぞれ違ったパターンで組み合わせた，計 16 種類の手法を評価した．EL 法 [8] は全ての手法に適用した．

[*2] `https://dumps.wikimedia.org/enwiki/`

表 1 削除されたキーの割合と充填率に関する結果.

キー削除率 (%)	0	10	20	30	40
充填率 (%)	100.0	89.4	78.9	68.5	58.1
キー削除率 (%)	50	60	70	80	90
充填率 (%)	47.9	37.8	27.8	18.1	8.7

4.2 実験結果と考察

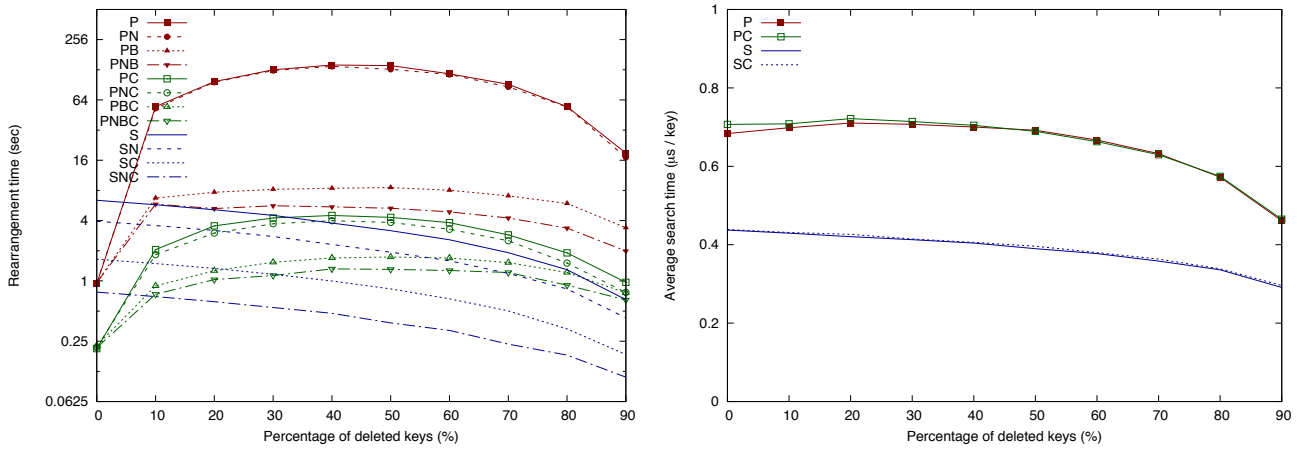
再構成時間に関する実験結果を図 5(a) に示し，検索時間に関する実験結果を図 5(b) に示す．再構成時間は，手法間で大きな差が生まれたため，図 5(a) の縦軸が 2 の対数軸となっていることに注意する．再構成時間に関して，静的再構築に BL 法を適用することによる変化は見られなかったため，これらの組み合わせに関する結果は省略している．検索時間に関して，NL 法と BL 法は関係がないため，並列化との組み合わせのみを結果として載せている．また，いずれの再構成法を用いた場合も，充填率は 100%近くにまで改善された．

まず，詰め直しによる再構成時間に着目すると，単純な再構成のとき，キー削除率 40–50%の地点で 140 秒も要している．NL 法を適用した場合も改善されず，枝刈挙が主な問題でないことがわかる．一方，BL 法を適用することで，この再構成時間は 8 秒にまで短縮されることから，いかにキャッシュ効率の悪さが従来の再構成法の問題点であったかがわかる．さらに，キャッシュ効率を改善することで NL 法が効率よく働き，BL 法と NL 法の組み合わせでは 5 秒で再構成が可能となる．並列化による高速化も確認され，NL 法，BL 法，並列化を組み合わせることで，再構成時間が 1 秒にまで短縮された．

次に，静的再構築による再構成時間に着目する．静的再構築では，全ての節点を辿る必要があるが，衝突回避による要素移動などが必要なくアルゴリズムが単純なため，詰め直しと比べ高速に再構成が可能という結果になった．特に，NL 法と並列化を組み合わせることで，登録キー数が 1,000 万件を超える状態であっても 1 秒未満で再構成が可能となる．再構成時間は主に節点数に依存するため，キー削除率が高い状態での再構成により適している．

最後に，詰め直しによる再構成法と静的再構築による再構成法の使い分けについて評価する．それぞれの最も性能が良かった場合の再構成時間を比較すると，キー削除率が 10%，すなわち充填率が約 90%の地点で再構成時間が逆転している．記憶効率を優先し，NL 法を適用しない場合で考えると，充填率が約 80%の地点で逆転している．そのため，辞書の構成などに応じて，充填率がある一定の値より高い場合は詰め直し，低い場合は静的再構築というように使い分けることで，高速な再構成がおこなえる．

一方で，再構成後の検索時間に着目すると，静的再構築によるキャッシュ効率化のおかげで，検索速度に明らかな差がでているのが伺える．キーの追加や削除においても，



(a) 再構成時間　　(b) 平均検索時間

図 5　再構成法に対する実験結果．凡例は，詰め直しによる再構成を P，静的再構築による再構成を S，NL 法を N，BL 法を B，並列化を C として手法の組み合わせを表している．

一度そのキーを検索する必要があるため，静的再構築によるキャッシュ効率化は，辞書としてのあらゆる操作の速度向上に繋がる．故に，充填率が低下していない状況においても，静的再構築による動的ダブル配列の再構成は重要であるといえる．また，定期的に静的再構築をおこなうという利用を考えたときに，並列化などによる高速化はより重要なものとなる．

5. おわりに

本稿では，実用における効率的な再構成法の検討と，新しい手法の提案，及び評価をおこなった．実験により，提案手法は従来より遥かに高速に再構成をおこなえることを示した．その再構成時間は 1,000 万件を超えるキーを登録した辞書においてもわずか 1 秒未満であり，実用において再構成時間が問題となることはない．また，静的再構築による再構成では，実行後の辞書の性能向上に繋がることを示し，充填率低下を改善する目的以外での再構成の必要性を示した．加えて，提案手法による高速化の有効性が確認された．今後の課題として，URL などの接頭辞に特徴を持ったキー集合に対して実験をおこなうなど，様々な場面に応じての評価を与えることがあげられる．

参考文献

[1] Knuth, D. E.: *The art of computer programming, 3: sorting and searching*, Addison Wesley, Redwood City, CA, USA, 2nd edition (1998).
[2] Aoe, J.: An efficient digital search algorithm by using a double-array structure, *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077 (1989).
[3] Yata, S., Oono, M., Morita, K., Fuketa, M., Sumitomo, T. and Aoe, J.: A compact static double-array keeping character codes, *Information Processing & Management*, Vol. 43, No. 1, pp. 237–247 (2007).
[4] Kanda, S., Fuketa, M., Morita, K. and Aoe, J.: A compression method of double-array structures using linear functions, *Knowledge and Information Systems*, Vol. 48, No. 1, pp. 55–80 (2016).
[5] Darts 0.32: Double-ARray Trie System, `http://chasen.org/~taku/software/darts/` (2008).
[6] Darts-clone 0.32g: A clone of Darts, `https://github.com/s-yata/darts-clone` (2011).
[7] Morita, K., Fuketa, M., Yamakawa, Y. and Aoe, J.: Fast insertion methods of a double-array structure, *Software: Practice and Experience*, Vol. 31, No. 1, pp. 43–65 (2001).
[8] 大野将樹，森田和宏，泓田正雄，青江順一：ダブル配列による自然言語辞書の高速更新法，第 11 回言語処理学会年次大会発表論文集，pp. 745–748 (2005).
[9] 矢田　晋，田村雅浩，森田和宏，泓田正雄，青江順一：ダブル配列による動的辞書の構成と評価，第 71 回情報処理学会全国大会講演論文集，pp. 1263–1264 (2009).
[10] 村山智也，望月久稔：ダブル配列構築の高速化を目的とした節点から遷移可能な遷移種の集合に基づく未使用要素の管理法，DEIM Forum 2016 論文集，pp. C6–4 (2016).
[11] Yoshinaga, N. and Kitsuregawa, M.: A self-adaptive classifier for efficient text-stream processing, In *Proc. of COLING*, pp. 1091–1102 (2014).
[12] Groonga 6.0.7: An open-source fulltext search engine and column store, `http://groonga.org/` (2016).
[13] Morita, K., Tanaka, A., Fuketa, M. and Aoe, J.: Implementation of update algorithms for a double-array structure, In *Proc. of IEEE SMC*, Vol. 1, pp. 494–499 (2001).
[14] Oono, M., Atlam, E.-S., Fuketa, M., Morita, K. and Aoe, J.: A fast and compact elimination method of empty elements from a double-array structure, *Software: Practice and Experience*, Vol. 33, No. 13, pp. 1229–1249 (2003).
[15] Yata, S., Oono, M., Morita, K., Fuketa, M. and Aoe, J.: An efficient deletion method for a minimal prefix double array, *Software: Practice and Experience*, Vol. 37, No. 5, pp. 523–534 (2007).
[16] 徳永拓之：日本語入力を支える技術 – 変わり続けるコンピュータと言葉の世界，chapter 3, 技術評論社 (2012).