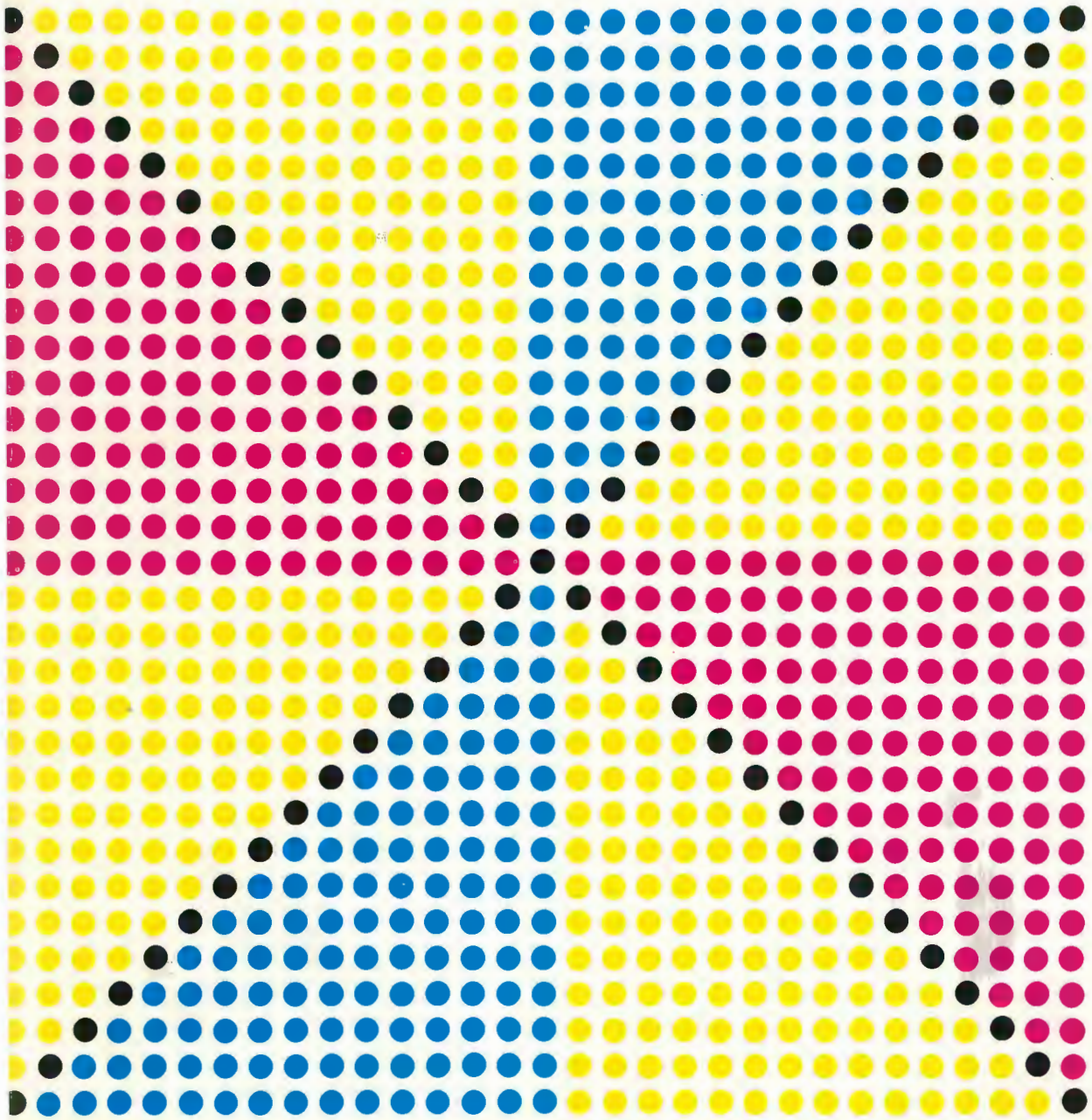
acm computing surveys '79

コンピュータ・サイエンス

コンピュータ・サイエンス誌 別冊

bit 11 1980

共立出版



目

次

COMPUTING SURVEYS

The Survey and Tutorial Journal of the ACM

Volume 11

広くゆきわたったB木

パデュー大学計算機科学科
D.Comer　訳　上田和紀

細　目　次

† ACM Computing Surveys, Vol. 11, No.2, pp. 121–138.
"The Ubiquitous B-Tree".

B木（B-tree）は事実上，ファイル編成のための一標準となった．B木を用いたファイル索引や，専用データベース・システムや，汎用アクセス法が，提案され実現している．この論文ではB木を概観し，なぜこんなに成功したのかを明らかにする．また B^+ 木をはじめとするB木の主な変形について，個々の実現方式の相対的な得失を対比させながら論じ，最後に，B木を用いた汎用アクセス法の例を示す．

はじめに

大型計算機システムで利用できる二次記憶機能によって，ユーザは，ファイルと呼ばれる大量の情報の集まりにデータを蓄えたり，そのデータを更新したり，読み出したりすることができる．しかし計算機があるデータ項目を処理するには，その前にそれを取り出して主記憶上に置かねばならない．そこで，計算機資源を上手に使うためには，ファイルをうまく編成（organize）し，検索処理の効率化をはかる必要がある．

よいファイル編成の選択は，行なわれる検索の種類によって決まる．検索の指令には大別して，下に例示されるような二種類のものがある．つまり，

順次的（sequential）：「わが社の従業員ファイルから，すべての従業員の名前と住所のリストを作成せよ」と

ランダム：「わが社の従業員ファイルから，従業員 J. Smith に関する情報を抽出せよ」とである．

ここで，各従業員に1枚ずつ割り当てられた紙挟みが，三段の引出しに収められているようなファイル・キャビネットを想像してみよう．引出しには“A-G”，“H-R”，“S-Z”などとラベルがつけられ，一方，紙挟みには従業員の姓を書いたラベルがつけられているかもしれない．順次的な検索要求のときは，検索者は紙挟みを1枚ずつ見ながら，ファイル全体を調べる必要がある．①方，ランダムな検索要求のときは，検索者は引出しと紙挟みについたラベルを見ながら1枚の紙挟みを選び出しさえすればよいのである．

計算機システムにおいては，ランダムアクセスされる大きなファイルには索引（index）というものがついていて，ファイル・キャビネットの引出しや紙挟みについたラベルと同様，欲しい項目を含んでいるファイル中の小部分を指し示すことによって検索の能率を上げている．図1は，ファイルとその索引を表わしたものである†．索引は，従業員の紙挟みについたラベルのようにファイルの中身と物理的にいっしょになっていることもあれば，引出しについたラベルのようにファイルの中身と物理的には分離されていることもある．もし索引づけされたファイルが大きければ，検索能率を上げるために，もうひとつの索引がもとの索引の上位につくられることもあり，さらにその上位へと積み重ねられることもある．こうしてできる索引の階層（hierarchy）は，さきの従業員ファイルのと同様のものである．従業員ファイルでは最上位の索引が引出しのラベルから成り，次のレベルの索引が紙挟みのラベルから成っていたのである．

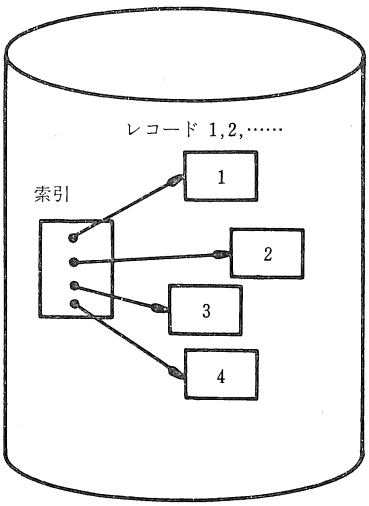


図1　二次記憶上のファイルとその索引

姓を索引項目とみなすことで形成されるような自然な階層は，計算機システムで使用した場合，いつも最高の性能をあげるとは限らない．通常は，ファイル内の個々の項目に一意的なキーが割り当てられ，すべての検索がキーを指定することによって行なわれる．たとえば個々の従業員に，その記録を識別するための一意的な従業員番号が割り当てられ，キャビネットの引出しには“A-G”

† （訳注）「ファイルとその索引」という表現がなされている場合，ファイルという語には単に「レコードの集まり」というぐらいの意味しかない．構造のほうにはほとんど興味がもたれていないのである．一方，索引は，そのファイルのレコードを構造化する役割をもってはいるが，ファイル内容にはまったく関与しない．

しかしもちろん，実際のファイル実体は，レコードを何らかの方法で構造化することで成立しているわけである．本論文中では，語「ファイル」は，この狭義・広義両方の意味で使われる．

などとラベルをつけるかわりに“0100-3000”というような従業員番号の範囲を用いるのである．

ファイルやその索引を編成するために数多くの方法が提案されてきており，Knuth［KNUT 73］がその基礎について概説を行なっている．どのひとつの方式をとってみてもすべての応用分野で最適というわけにはいかないが，そのなかでB木と呼ばれる，ファイルとその索引の編成手法が広く使われるようになった．B木は事実上，データベース・システムにおける索引のための標準的な編成となっている．この論文は，B木のことを耳にしたことがあって，その解説や，さらに勉強をすすめるための方向づけを求めている計算機のプロを念頭において，B木のいくつかの変形を比較し，特にその中で B^+ 木がなぜ普及したかを明らかにしている．またこの論文はB木に関する文献について，教科書に触れられていない最近の論文も含めて概観している．さらに，B木をもとにした汎用のファイル・アクセス法についても論じている．

われわれの議論の出発点は，二分探索木(binary search tree) と呼ばれる内部記憶構造である．なかでも，安い検索費用が保証されているということで，均衡二分探索木(balanced——)からとりかかる．二分探索木や他の内部記憶機構を概観したい読者は，SEVE 74 や NIEV 74 を参照されたい．NIEV 74 には，ここでの議論を通じて使われる木 (tree)，節 (node)，辺 (edge)，根 (root)，路 (path)，葉 (leaf) といったグラフ理論の用語† も解説されている．

この序説の残りでは，検索処理のモデルを与え，考えるべきファイル操作の大筋を示す．1. では Bayer と McCreight によって提案された基本B木を，項目の挿入，削除，探索の手法とともに紹介する．次に，それぞれの操作について 2. で費用を調べ，順処理が高くつくことを導き出す．実現方法を変えてみると費用が安くなるということはしばしばあるものだが，3. ではこの目的で開発されたB木の変形を示す．変形をいろいろ紹介してから 4. では複数ユーザ環境においてB木を保守する問題にふれ，並行性と安全性の問題に対する解の大筋を示す．最後に 5. でB木をもとにした IBM の汎用のファイル・アクセス法を紹介する．

ファイルに対する操作

この論文においては，**ファイル**とは n 個の**レコード**の集まりであり，個々のレコードは $r_i=(k_i, \alpha_i)$ なる形をしていると考える．ここで k_i は i 番目のレコードの**キー**，α_i は**関連情報** (associated information) と呼ばれる．たとえば従業員ファイルならば，レコードのキーは5桁の従業員番号で，関連情報は従業員の名前，住所，給与，扶養家族数から成っていたりするのである．

われわれは，キー k_i はレコード r_i を一意的に識別するものと仮定する．さらにわれわれは，キーは関連情報よりはるかに短いが，全レコードのキーを集めると主記憶に入りきらぬほどの量になることも仮定する．これらの仮定は，もしキーを用いてレコードをランダムに検索しようとするならば，能率向上のために索引を作成するのが有利であるということを意味している．ただ，キー全体が一度に主記憶に入りきらないのだから，索引自身は外部に置かねばならない．最後にわれわれは，ファイルの**キーによる順序** (キー順，key-sequence order) に言及できるように，キーがアルファベット順のような自然の順序をもっていることを仮定する．

ユーザはファイルを相手に**やりとり** (transaction) を行ない，レコードを挿入したり，削除したり，検索したり更新したりする．これに加えユーザはたびたび，与えられた点から始めて，キー順に従って順次的にファイルを処理する．その開始点はファイルの先頭である場合が最も多い．このようなやりとりを支える**基本操作**の組は次のようなものから成る．

† (訳注) NIEV 74 にアクセスしにくい読者のために若干の説明を補っておこう．ここで興味のもたれている木は，グラフ理論において最も一般的に定義されている木ではなく，「根とよばれる1個の節が，0個以上の，順序づけられた，共通部分をもたない部分木(これがまた木をなす)をしたがえたもの」として定義される「順序木 (ordered tree)」である．したがって，これは概念的には有向グラフの部分集合なのだが，実際にはあとで出てくるように，木を（根に向かって）さかのぼるといった操作も行なわれる．葉とは部分木をもたない節のことである．路とは，ある節から始まり他の節へいたるまでの，節と辺を交互に並べた系列のことである．この論文中のほとんどの箇所では，路といえば有向路（辺の方向がそろったもの）を指しているが，そうでない箇所もある．

なお，ここでは触れられていないが，深さ，あるいはレベルといった語も多用されている．これらの語の定義は人によってまちまちなので注意を要するのだが，ここでは以下のように定められている（p. 28†3参照）．

① 節の深さ（あるいはレベル）とは，根からその節までの路に含まれる辺の数である．ただし，根の深さは0とする．

② 木の深さとは，その木の葉の深さのことである．

なお，「高い（上位の）レベル」といういい方が論文中に出現するが，ここでの高低の概念は，黒板に木を（自然界に存在する木とは逆の枝ぶりになるように）描いたときの上下関係に一致するものである．レベル番号が小さいほど高レベルであるという関係に注意されたい．

挿入（insert）：k_i が一意的であることを検査して，新しいコード（k_i, α_i）を追加する．

削除（delete）：k_i を与えて，レコード（k_i, α_i）を取り除く．

検索（find）：k_i を与えて α_i を取り出す．

「次」（next）：α_i が取り出されたばかりだとするとき，α_{i+1} を取り出す（すなわち，ファイルを順次的に処理する）．

与えられたファイル編成に対して，索引の保守やこれらの個々の操作の実行に関連する費用というものが存在する．検索を速くする目的で索引をつけるくらいだから，通常は処理時間が主要な費用尺度とされる．現在のハードウェア技術では，二次記憶のアクセスに要する時間が，データ処理に必要な合計時間の主要部分を占めている．さらに，たいがいのランダム・アクセス装置は1回の読出し操作で，決まった量のデータを転送するから，必要な合計時間は，読出しの回数に正比例することになる．このことより，二次記憶へのアクセス回数が，索引の方式を評価する際の妥当な費用尺度として役立つのである．なお，ほかの，これよりも重要性の低い費用尺度としては，いったん主記憶に置かれたデータの処理に要する時間，二次記憶の空間利用効率，索引のために必要な空間と関連情報のために必要な空間の比，といったものがある．

1. 基本B木（Basic B-tree）

B木は，短いけれども重要な歴史をもっている．1960年代の終りに計算機メーカと，独立した研究グループとが競いあって，計算機用の汎用ファイル・システムといわゆる「アクセス法」を開発した．Sperry Univac 社では H. Chiat や M. Schwartz らが（Case Western Reserve 大学と共同で），後述するB木による手法に関連したやり方で検索や挿入を行なうシステムを開発し，実現した．これとは独立に，B. Cole, S. Radcliffe, M. Kaufmann らは Control Data 社（CDC）で（Stanford 大学と共同で），同様のシステムを開発した．そののち R. Bayer と E. McCreight が Boeing 科学研究所において，前節に定義した操作の大部分が比較的安い費用でできる外部索引機構を提案し，B木[†1] と名づけた [BAYE 72].

この節では，基本B木のデータ構造と保守算法とを，節から2本以上の路が出ることのない二分探索木を一般化したものとして紹介し，次節では個々の操作の費用について論じる．なお，一般的な概説はほかに HORO 76, KNUT 73, WIRT 76 にもある．

二分探索木において，ある節で選ばれる分枝は**問合せキー**（query key）とその節にしまわれたキーとの比較の結果によって決まっていたことを思い出していただきたい．もし問合せキーの値が格納されたキーの値より小さければ左の枝がとられ，問合せキーのほうが大きければ右の枝がとられる．**図2**は，従業員番号の格納に用いられているそのような二分木の一部と，"15" という問合せに対してとられる路を示すものである[†2]．

さて次に，**図3**に示されているような，個々の節にふたつずつキーの入った，修正された探索木を考えよう．探索は，それぞれの節でみっつの枝のうちのひとつを選ぶことにより進行する．図3では，問合せ値15が42よりも小さいので，根においては最も左の枝がとられる．これが

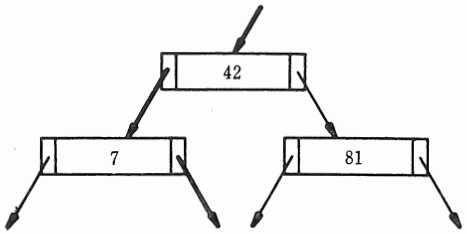


図 2　従業員番号の二分探索木の一部．太線は，問合せ "15" に対してとられる路を示す

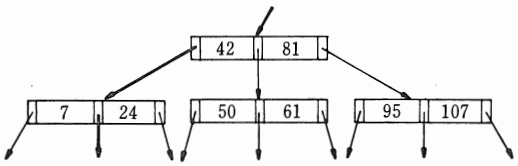


図 3　個々の節に2個のキーと3本のポインタをもった探索木．太線は問合せ "15" に対してとられる路を示す

---

†1（原注）「B木」の語源については，この著者たちは何も説明していない．われわれの考えるところでは，"balanced（均衡した）", "broad（幅広い）", "bushy（灌木のような）" などがあてはまりそうである．また，"B" は Boeing を表わしているという人もいる．しかし，Bayer の貢献を考えると，B木を "Bayer-tree" の略とみなすのが適当であるように思える．

（以下訳注）訳者も，B-tree の語語をB木にするかB氏木にするかでずいぶん悩んだが，この原注にあるように真の語源が不明であること，および B* 氏木やB氏* 木はどうもいただけないということから，B木，B* 木，B+ 木などの語を用いることにした．

†2（訳注）図2以下，図17あたりまでの図には，キーは描かれているが関連情報が描かれていない．議論に無関係ということで略されているわけだが，実際にはキーのそばに関連情報そのもの，あるいはそれへのポインタが格納されていると考えるのである．

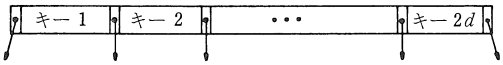


図 4 $2d$ 個のキーと $2d+1$ 本のポインタをもつ，位数 d のB木の節

42 と 81 の間の値の問合せならば中央の枝が選ばれ，81 より大きな値の問合せならば最も右の枝に従うことになる．この決定手続きは，厳密な一致が起きる(探索成功)か，葉にいたる（失敗）まで，各節において繰り返される．

一般に，**位数 d のB木**は，図4に示すように最大 $2d$ 個のキーと $2d+1$ 本のポインタをもつ†1．実際にはキーの数は節によって異なりうるが，各節は少なくとも d 個のキーと $d+1$ 本のポインタをもたねばならない．その結果，個々の節は少なくとも 1/2 は埋まっている†2．通常の実現方法では，節は索引ファイルのひとつのレコードをなし，$2d$ 個のキーと $2d+1$ 個†3のポインタを収納できるだけの固定された長さをもち，さらに節のなかにいくつのキーがちゃんとした情報として入っているかを示す付加情報を蓄えている．

ふつう，キーをたくさんもった大きな節は主記憶に保持しておくことができず，検索の際は毎回二次記憶をアクセスする必要がある．あとでわれわれは，節に2個以上のキーをもつと，検索・挿入・削除操作の費用がわれわれの費用基準の下でいかに安くなるかを調べるであろう．

均衡化 (Balancing)

B木の美しさは，木を常に均衡させたままレコードの挿入や削除を行なう方法にある．二分探索木の場合がそうであるように，ファイルにレコードをランダムに挿入すると木が不均衡になることがある．不均衡木とは図5(a)に示すようなもので，長い路や短い路をもつが，一方図5(b)に示すような均衡木では，同じ深さの所にすべての葉がある．直観的には，B木は図6に示されるような形をもっている．n 個のキーをもつB木の最長路には，d をB木の位数として，最大およそ $\log_d n$ 個†4の節がある．n レコードのファイルの索引をなす不均衡木において検索操作を行なうと，n 個の節を訪れることになるかもしれないが，そのようなファイルの索引のなす位数 d のB木においては，けっして $1+\log_d n$ 個†5より多くの節を訪れることはない．節への訪問は毎回二次記憶へのアクセスを要するので，木を均衡させておくことは大きな節約につながる可能性がある．木の均衡化については，多くの方式が提案されてきた（実例については NIEV 74, FOST 65, KARL 76 を見よ）．どの方式でも，均衡化を行なうには多少の計算時間を要するから，検索操作の間の節約は均衡化自体の費用を上回るものでなければならない．その点，B木の均衡化方式は木の変更を葉から根への1回の走査に限定しているので，「とめどもない(runaway)」オーバーヘッドをもたらすことがない．さらにB木の均衡化機構は，均衡化の費用を安くす

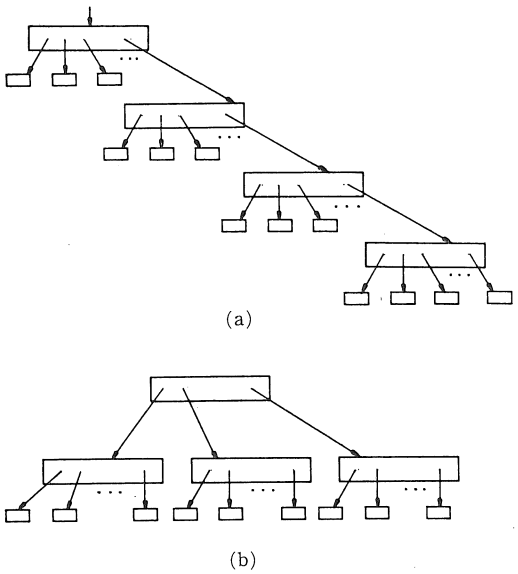


図 5 (a) 長い路を多数もった不均衡木と，(b) 葉にいたるすべての路の長さがまったく等しい均衡木

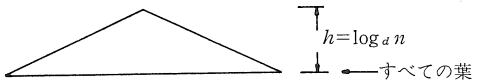


図 6 n レコードのファイルを索引づける，位数 d のB木の輪郭

†1 (訳注) ここでの位数の定義は Bayer 流のものである．このほかに，$2d+1$(ポインタの最大本数) を位数とする，Knuth 流 [KNUT 73] の定義もある．

†2 (訳注) これは根以外の節の話であり，根については 2. にあるように，最低1個のキーと2本のポインタとがあればよいことになっている．

†3 (訳注) 原文では $2d$ 個となっている．ミスプリであろう．

†4 (訳注) これは，d が大きく，さらに n が d に比べて十分大きいときの漸近的な評価である．本当は，対数の底は $d+1$ という形をとるのだが，B木を二次記憶上の外部索引として実際に使うときには d を数十といった値（あるいはそれ以上）にすることが多いから，d でも $d+1$ でもたいして変わらない，というわけである．しかし一方，後出の 2-3 木や二分B木のように，$d=1$ とする使い方も立派に存在し，そのような場合にはこの評価式はまったく役立たずになる．

†5 (訳注) 「1+」がついているのは，「けっして」という言葉に対応してであろうか？

るために余分な記憶域を用いている（たぶん，検索時間に比べて二次記憶域は安価なものである）．そういうわけで，B木は均衡木の方式の利点を備える一方で，時間のかかる保守作業のいくらかをしないで済ませているのである．

挿入（Insertion）

木の均衡が挿入の間どのように保たれるかを見るために，図 7(a)の位数2のB木を考えてみよう．位数 d のB木の節が d 個ないし $2d$ 個のキーをもつことから，この例における節は2個から4個のキーをもつ．また図には描かれていないが，現在のキーの数を憶えておくために，個々の節に何か標識がついていなければならない．新しいキーの挿入には二段階の処理を要する．まず，挿入を行なうべき適当な葉を探し出すために根から検索処理が進行する．続いて挿入が行なわれ，葉から根のほうへと進行する手続きによって均衡が回復される．図7(a)を例にとってみよう．キー “57” を挿入しようとすると，左から四つめの葉で検索が不成功におわることがわかる．しかしその葉にはもうひとつキーを収容できるから，単に新たなキーが挿入され，図 7(b)のような木になるだけである．ところが，キー “72” を挿入しようとすると，めんどうな事態が生じる．というのは，挿入に適した葉がすでに一杯だからである．すでに一杯になっている節にキーを挿入する必要のあるときは，いつも**分割**（split）が起こり，今の場合節は図8に示すように分けられる．$2d+1$ 個のキーのうち，小さいほうから d 個がひとつの節に置かれ，大きいほうから d 個がもうひとつの節に置かれ，残った値は親の節に昇進して分離値として役立てられるのである．さて通常の場合は，親の節が追加キーを収容して，挿入処理が終了する．しかし，もし親の節もたまたま一杯であったならば，同様の分割処理が再び行なわれる．最悪の場合，分割ははるばる根まで伝播し，木は1レベルだけ高さを増す．実は，B木は根における分割によってのみ高さを増すのである．

削除（Deletion）

B木における削除もまた，正しい節を見つけ出すため

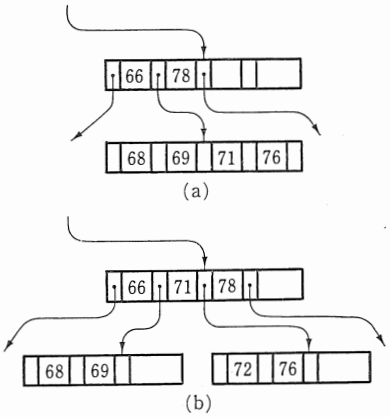


図 8　(a) B木の葉とその祖先，および(b) それにキー“72”を挿入したあとの部分木．個々の節は2ないし4（d ないし $2d$）個のキーをもつ

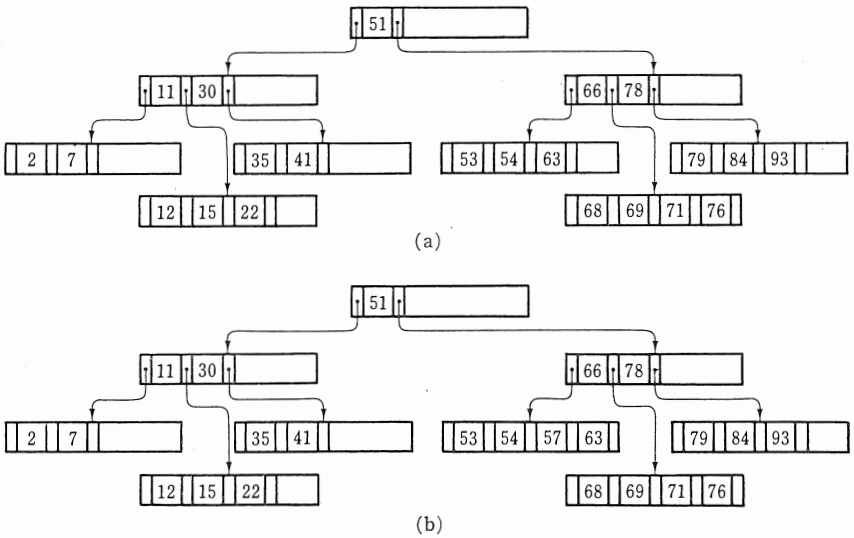


図 7　(a) 位数2のB木と，(b) それにキー “57” を挿入したあとの木．根のキーの数は，そのB木の位数 d より少なくてもよいことに注意されたい．それ以外の節はすべて，少なくとも d 個以上のキーをもつ

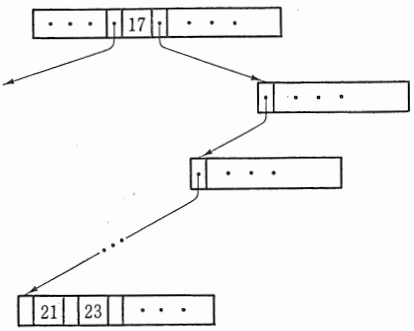


図 9 キー "17" を削除するときは，キー順で次にあたるキー "21" を探し出して，それを空いた場所に移してこなければならない．キー順で次にあたるキーは，いま空いた場所の右側のポインタで示される部分木の，最も左の葉に必ず存在する

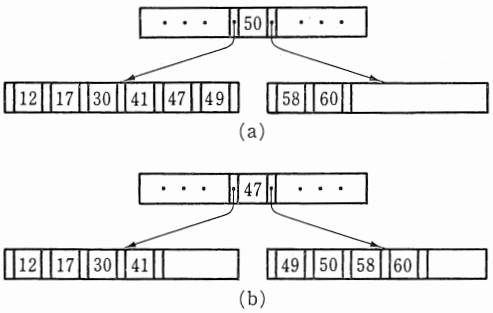


図 10 隣りの節との間でキーを再配分する前(a)と後(b)のB木の一部．分離キー "50" の最終的な位置に注意．再配分によって節の大きさを等しくすると，ひきつづく削除においてアンダーフローが発生しにくくなるという利点がある

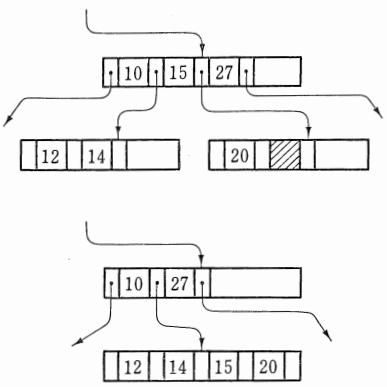


図 11 (a) 連結を引き起こす削除と，(b) 再均衡化された木

† (訳注)「不要になり」とは，「分離値として働く必要はなくなり」ということである．

に検索操作を要する．検索がすんだあとは，ふたつの可能性がある．つまり削除すべきキーが葉にあったか，葉でない節にあったかである．葉でない節における削除では，隣接したキーを見つけ，それがちゃんと仕事にありつけるように，空いた場所に移してこなければならない．キー順で隣接しているキーを見つけ出すには，いま空いた区画の右側の部分木の最も左の葉を探すだけでよい．二分探索木の場合と同様，必要な値は必ず葉に存在する．図9がこれらの関係を示している．

さて，いったん空き区画が葉へ「移されて」きたら，われわれはキーが少なくとも d 個残っているか検査しなければならない．d 個未満のキーしか葉になかった場合は「**アンダーフロー**」が起きたといわれ，キーの再配分が必要となる．均衡（それと個々の節は少なくとも d 個のキーをもつというB木の性質）を取りもどすには，ただひとつのキーがあればよい．そしてそれは隣りの葉から借りてくることによって得られる．しかしこの操作は少なくとも2回の，二次記憶へのアクセスを要するのだから，隣りあったふたつの節の間で残されたキーを均等に分けあって，同一の節からの連続的な削除の費用を安くするというのが，よりましな再配分法であろう．再配分は，図 10 に例示されている．

もちろん，隣りどうしでのキーの配分がうまくゆくのは，配分するキーが少なくとも $2d$ 個あるときだけである．もし $2d$ 個未満のキーしか残っていなければ，**連結** (concatenation) が起きなければならない．連結作業の間にキーは単にどちらかの節に統合され，もうひとつの節は捨てられる（連結は分割の逆であることに注意されたい）．ひとつの節しか残らないので，先祖の節にあってふたつの節を分離していたキーはもはや不要になり†，これもまたただひとつ残った葉につけ加えられる．図11は，連結と，分離キーの最終的配置の例を示すものである．

ある節が，その子のうちの二者の連結によって分離キーを失うとき，その節もまたアンダーフローを起こし，隣りの節のうちのひとつからキーをもらい受けなければならなくなるかもしれない．すなわち，連結処理はすぐ上位のレベルにおける連結を引き起こすかもしれず，それが繰り返されて根にいたるかもしれない．最終的に根の子供たちが連結されてしまうならば，それが新たな根を形成し，B木の高さは1だけ減る．

なお，挿入と削除の算法は BAYE 72 にあり，Pascal でプログラムされた簡単な例が Wirth [WIRT 76] によって与えられている．

2. 諸操作の費用

B木の節を訪れるには二次記憶へのアクセスを要するので，操作の間に訪れた節の個数が，その費用の尺度を与えてくれる．Bayer と McCreight [BAYE 72] は，挿入，削除，検索の費用の厳密な解析を行なっている．彼らはまた，理論的な費用の限界を実際の装置に関連づける，広範囲にわたる実験の結果を与えている．Knuth [KNUT 73] も，やや異なった定義を用いて，B木における操作費用の限界を導いている．次節では費用の漸近的な限界について，簡単な解説を行なう．

検索費用

まず，検索操作の費用を考えよう†1．根を例外として，B木にはひとつの節につき d 個ないし $2d$ 個のキーがあるから，個々の節は少なくとも $d+1$ 個†2の子をもっている．また根は少なくとも2個の子をもつ．そこで，深さ†3 0, 1, 2, 3, …の節の個数は少なくともそれぞれ 1, 2, $2(d+1)$, $2(d+1)^2, \cdots$†4 でなければならない．すべての葉は深さ h の所にあるから，全体では節は

$$N=1+\sum_{i=1}^{h}2(d+1)^{i-1}=1+2\cdot\frac{(d+1)^h-1}{d}$$ †5

個以上あり，根以外は†6 それぞれが d 個以上のキーをもっている．したがって，合計 n 個のキーをもつ木の高さは

$$1+(N-1)d=2(d+1)^h-1\leq n$$ †7

†1（訳注） ここからの解析は，原文では漸近的な評価に基づいているということであったが，漸近的というよりはむしろ不正確なものであり，しかも訳注 p. 25†4でも述べたように $d=1$ のような場合にまったく適用できないものであったので，大幅に手直しした．訳文中の式は厳密に成り立つものになっており，しかも原文のものより明快である．

†2（訳注） 原文では d 個．

†3（原注） 木の根は深さ0にあり，深さ $i-1$ にある節の息子は深さ i にある．（p. 23†参照）

†4（訳注） 原文では $2, 2d, 2d^2, 2d^3, \cdots$．

†5（訳注） 原文では $\sum_{i=0}^{h}d^i=\frac{d^h-1}{2d-1}$．左辺の係数“2”の脱落と，右辺の分母にある“2”のミスプリを考慮してもなお，これは等式として成立しない．

†6（訳注） 訳者追加．

†7（訳注） 原文では $2d(d^h-1)/(d-1)\leq n$．

†8（訳注） 原文では $2d^h\leq n+1$．

†9（訳注） 原文では $h\leq\log_d\frac{n+1}{2}$．なお，h は「根から葉へいたる路上の節の数」よりはひとつだけ少ないことに注意．

†10（訳注） 訳注 p. 25†4参照．

表1　いろいろな節とファイルの大きさにおける，アクセスされる節の数の最悪の場合の上限

Node size ↓ \ File size (records) →	10^3	10^4	10^5	10^6	10^7
10	3	4	5	6	7
50	2	3	3	4	4
100	2	2	3	3	4
150	2	2	3	3	4

で押さえられ，簡単な計算によって

$$2(d+1)^h\leq n+1$$ †8

すなわち

$$h\leq\log_{d+1}\frac{n+1}{2}$$ †9

が示される．このように，検索操作の処理の費用はファイルの大きさの対数に比例して増大する．

表1は，大きなファイルに対しても，対数的費用というものがいかに妥当なものでありうるかを示している．100万レコードのファイルの索引をなす位数50のB木が，最悪の場合でもわずか4回のディスク・アクセスで探索できるのである．あとでわれわれは，この見積りは過大であり，簡単な実現技法によって最悪の場合の費用を3に，平均費用はそれよりも小さくできることを確かめるであろう．

Aho ら [AHO 74] は，B木の検索費用について別の見通しを与えている．彼らは，探索が個々の節における比較に基づいているような決定木（decision-tree）モデルの演算については，これよりも漸近的に高速な算法はあみだせないことを示している．もちろんこのモデルは，ちらし（hashing）[MAUE 75] のようなある種の手法を除外してはいる．しかしそれでもなおB木は，実用的と理論的の双方の意味で安い検索費用を示しているのである．

挿入・削除費用

挿入あるいは削除の操作は木を上向きにさかのぼるかもしれないから，検索費用のほかに余分な二次記憶へのアクセスを要することがある．全体では費用は最大2倍となるが，木の高さが依然としてその費用の式を支配している．したがって，n レコードのファイルのための位数 d の B 木では，挿入や削除に最悪の場合で $\log_d n$†10 に比例する時間がかかる．

いまや，節に多数のキーを収容することの利点は明白になったはずである．分岐係数 d を増やせば対数の底が

大きくなり，検索，挿入，削除操作の費用が減るのである．しかし，節の大きさには実用上の限界がある．たいがいのハードウェアは，1回の二次記憶へのアクセスで転送できるデータ量を制限している．また，われわれの費用の式は，転送データの大きさが増すにつれて大きくなってくる定数係数を書いていない．さらに，個々の装置は，大量の空間のむだ遣いを防ぐためにはこれだけデータを収容しなければならないという，ある決まったトラック長というものをもっている．そこで実際には，最適な節の大きさは，システムの特性とファイルが割り付けられる装置の特性とに非常に大きく依存するのである．

Bayer と McCreight [BAYE 72] は，節の大きさを（ディスクやドラムの）回転待ち時間，転送速度，およびキーの大きさに基づいて選ぶための，おおよその指針を与えている．彼らの実験は，そのモデルの最適値で実際にもうまくゆくことを証明している．

順 処 理

これまでは，われわれはキーを指定することによって行なわれるランダムな処理を考えてきた．しかし，ユーザとしてはファイルを順次的なものとみなしたくなることもたびたびあり，すべてのレコードをキー順に処理するために「次」という操作を用いる．実際，B木に対するひとつの代案であるいわゆる索引順アクセス法 (Indexed Sequential Access Method, ISAM) [GHOS 69] は，順処理が非常に頻繁に起きることを仮定している．

残念ながらB木は，順処理の環境ではあまりうまくないかもしれない．単純な通りがけ順 (中順，inorder)†1 の木のなぞり (tree walk) [KNUT 68] によってすべてのキーを順に取り出すことができるが，それには主記憶に少なくとも $h=\log_{d+1}(n+1)$†2 個の節を収容するだけの空間がいる．というのは，節の二度読みを避けるために，根からの路上の節を棚 (stack) に積んでおくからである．おまけに「次」の操作を行なうには，目的のキーに到達するまでに，いくつかの節を通りながら路をたどらなければならないかもしれない．たとえば，最小のキーは最も左の葉の中に存在するが，これを見つけるには図12に示すように，根から葉にいたる路上のすべての節をアクセスしなければならない．

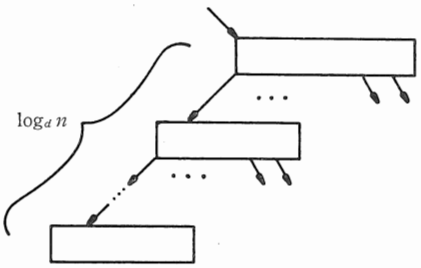


図 12 B木の最も左の葉にある最小キーの探索．それに行きつくには $\log_d n$ 回のアクセスを要する

「次」という操作の費用を改善するために何かできないだろうか．この疑問やその他の疑問に対する答は，次章において「B^+ 木」の話題のところで与えられるであろう．

3. B木の変形

たいがいのファイル編成の場合と同様，B木にも多くの変形がある．Bayer と McCreight [BAYE 72] は彼らの最初の論文で，いくつかの実現方式の代案を示唆している．たとえば，削除の結果生じるアンダーフロー条件は，(必要な数のキーが得られるならば) 隣りの節からキーを融通してもらうことによって連結を行なわずに対処できる．同じ戦略をオーバーフロー条件にも適用すると，分割を遅らせて，それに伴うオーバーヘッドをなくすことができる．つまり，節が一杯になったからといってすぐ分割してしまわず，キーを単に隣りの節に分けてやって，両隣りが一杯になったときだけ節を割るのである．

ほかのB木の変形は，二次的な費用の改善に意を注いでいる．Clampet [CLAM 64] は，二次記憶からいったん取り出されたあとの節の処理の費用を考察している．彼は，正しい子へのポインタを見つけ出すのに順探索ではなく二分探索を用いることを示唆している．Knuth [KNUT 73] は，節が大きければ二分探索が有用だが，小さな節に対しては順探索が最適であると指摘している．しかし内部探索を順探索と二分探索に限定する理由はない．KNUT73 にある技法のなかに，いくらでも使えそうなものがある．とくに Maruyama と Smith [MARU 77] は，彼らが平方根探索†3と呼んでいる補外の技法について言及している．

†1 (訳注) 原文では「行きがけ順(先順，preorder)」となっている．

†2 (訳注) 原文では $h=\log_d(n+1)$．

†3 (訳注) これは飛越し探索 (jump searching) [SCHN 78] と呼ばれる技法のうち最も単純なものである．整列されたキーの列をいくつかのブロックに分割しておき，まず探索キーと各ブロックの最大キーとを比較しながら，求めるキーを含んだブロックを探し出す．ブロックが見つかったらあとはそのなかを順探索するのである．キーの総数が N であるとき，ブロックの大きさを $\sqrt{N}$ にすると，平均 $\sqrt{N}$ 回の比較で探索ができるので平方根探索と呼ばれる．

Ghosh と Senko［GHOS 69］は，ファイルの索引の作成を一般的に論じたなかで，二次記憶へのアクセスを省くために補間探索†1を利用することを考えている．そこで与えられた解析はB木をも包括するものであり，葉のすぐ上の何段かの索引レベルをやめてしまうのが費用の面で有効であることが示されている．ただその場合，検索はいくつかの候補の葉をあげたところで終了するので，正しい葉はキーの値と分布に基づく「見積り」によって見つけることになる．見積りによって誤った葉を選んだときは，順探索を行なえばよい．見積りのうちのいくつかははずれるかもしれないが，この方法は平均的には引き合うものである．

Knuth［KNUT 73］は，それぞれの深さで異なった「位数」をもつようなB木の変形を示唆している．その動機の一部は，葉の節におけるポインタ用の場所はむだで，省くべきであるという彼の観察からきている．根に対して異なった形を与える（根は，ほかの節に比べたらめったに一杯にならない）のも，意味のあることである．しかし，この実現方式の保守費用は，利益に照らし合わせてみて高価であるように思える．というのはとくに，二次記憶というものが安価で，かつ固定長の節によく適合するものであるからである．

B* 木

おそらく，B木の文献において最も誤用されている語はB* 木であろう†2,3．実は，Knuth［KNUT 73］は B* 木を，根以外の†4 それぞれの節が（1/2 ではなくて）少なくとも 2/3 は埋まっているようなB木として定義しているのである．B* 木における挿入では，二つの兄弟の節が一杯になるまで分割を遅らせるのに，局所的再配分の技法を用いる．そののちに，二つの節が三つに分割され，それぞれが 2/3 だけ埋まった状態になるのである†5．この手法は少なくとも 66%†6 の記憶効率を保証する一方で，保守の算法に対してはほどほどの調整しか要しない．記憶効率を高めると，できる木の高さが低くなるために検索が高速化されるという副次的効果があることも，指摘しておかねばなるまい．

B* 木という語は，やはり Knuth によって示唆された（［KNUT 73, WEDE 77, BAYE 77］参照），もうひとつの大変有名なB木の変形に対してもたびたび用いられている．混乱を避けるためにわれわれは，この Knuth の無名の実現方式に対しては B+ 木という語を用いることにしよう．

B+ 木

B+ 木では，すべてのキーが葉の中にある†7．B木として編成されている上位のレベルは，索引，つまり索引部とキー部の高速な探索を可能にするための道路地図だけから成っている．**図 13** は，索引部とキー部の論理的な分離を示すものである．当然，索引部の節と葉の節とは異なる形式でよく，さらに異なる大きさであってもかまわない．とくに，葉の節は通常，図に示されているように，左から右へつなぎ(link) で結ばれている．この，葉のつなぎ並び（linked list）はシーケンスセットとよばれる．シーケンスセットのつなぎは順処理を容易にする．

B+ 木の良さを十分に評価するには，索引とシーケンスセットを独立にもつことに含まれている深い意味を理解しなければならない．しばらくの間，検索操作を考えてみよう．探索は B+ 木の根から始まり，索引を通って葉にいたる．すべてのキーが葉にあるのだから，路が正しい葉に向かっている限り，検索の途中でどんな値に出

---

†1（訳注）　これは，整列されたキーの列の両端のキー値と，キーの分布状態とから，求めるキーの位置を補間法で推定するというものである．推定がはずれていたら，推定位置で区切られる二本の部分のうち，求めるキーが含まれているほうで同じことを繰り返す．キーが一様に分布しているときは，線形補間によって，$O(\log \log N)$（Nはキー列の長さ）の手間で探索ができることが示されている［PERL 78］．ただ本論文で述べてある文式では，補間による推定は一度しか行なわないようである．

†2（訳注）　実際，B* 木という語は，後出の B+ 木と同義に用いられたり（Bayer 氏もこの「誤用」派のひとりである），B+ 木と B* 木の複合形に用いられたりしていることが多いから，注意を要する．

†3（原注）　おもしろいケースは，B* と名づけられた木探索算法を指す，"B* tree search algorithm"［BERL 78］である．

†4（訳注）　訳者追加．

†5（訳注）　Knuth の定義では常に 2/3 以上節を埋めておかねばならないことになっている．しかし Bayer の原論文［BAYE 72］では，削除によりキーが減る場合には，記憶効率が 1/2 まで下がりうることになっている．実は，特定の箇所に挿入と削除が集中的に起こるような場合には，節の記憶効率が 2/3 を割った時点ですぐ連結を行なうよりも，Bayer の方式のようにヒステリシスをもたせたほうが，分割と連結の回数が減るという点で望ましい．

†6（訳注）　この数字も，実用的にはまったくこれで差しつかえないが，根の存在を考えるときは，漸近的な数字として解さなければならない．なお，B* 木における根には，2 個以上 $2\left\lfloor \frac{4d}{3} \right\rfloor + 1$ 個までのキーを蓄えることができる．

†7（訳注）　これは，「すべての関連情報も葉のなかにある」ということを意味する．葉でない節にあるものは，関連情報を伴わない仮のキーにすぎないのである．

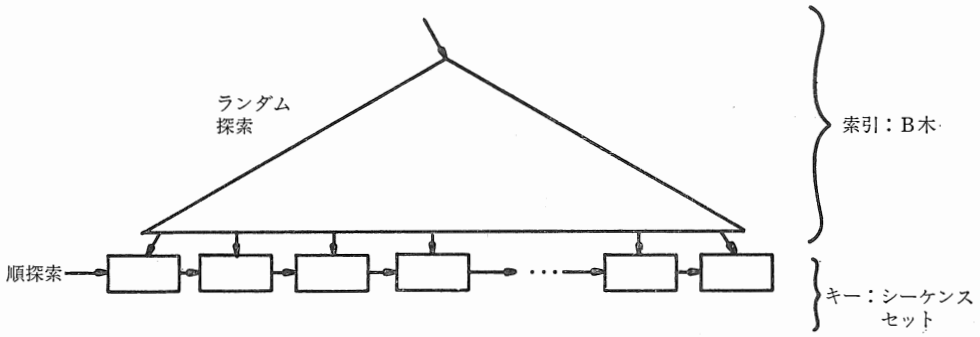


図 13 索引部とキー部を別々にもった B^+ 木．「キーによる」操作はB木の場合のように根から始まり，順処理は最も左の葉から始まる

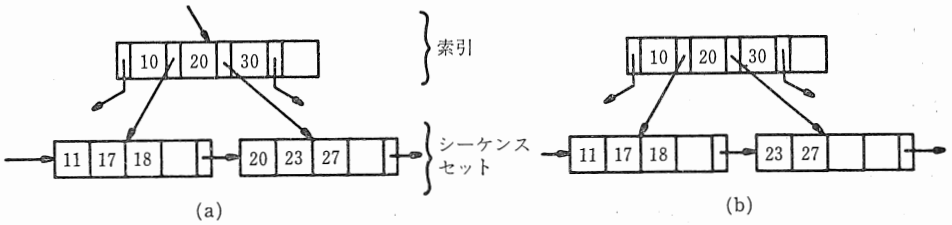


図 14 （a）B^+ 木と，（b）それからキー "20" を削除したあとの B^+ 木．削除後もなお，キー "20" は索引部において，分離値として役立てられる

会うかということは問題にならない．

B^+ 木における削除では，キーでない値を分離値として索引部に残しておけるということが，処理を単純化している．消すべきキーは必ず葉にあるから，その削除は簡単である．葉が少なくとも半分埋まっている限り，そのキーの写しが索引部に持ち込まれていたとしても，索引を変更する必要がない．図 14 は，消去されたキーの写しが，いかにして正しい葉へ検索処理を導くことができるかを示している．もちろん，もしアンダーフロー条件が生じたならば，再配分や連結の手続きは葉におけるのと同様，索引においても，調整用の値を要求するかもしれない．

B^+ 木における挿入や検索の操作は，B木における挿入や検索の操作とほとんど同じように処理される．ただ葉がふたつに割れるときは，中央のキーを昇進させる代わりにそのキーの写し（copy）を昇進させ，実際のキーは右側の葉に残すようにする．検索処理は，索引のなかのキーが問合せ値と等しくなっても探索が中止されないという点で，B木のそれと異なっている．その代わりに，右側の最も近いポインタに従って，はるばる葉にいたるまで探索が進められるのである．

われわれは，B木が安い検索・挿入・削除操作を支援していながら，「次」をとる操作には $\log_d n$ 回の二次記憶へのアクセスを要するかもしれないということを見てきた．B^+ 木の実現方式は，キーによる操作の対数的な費用特性を保ちながら，「次」という操作をとり行なうのに高々1回の二次記憶へのアクセスしか要しないという利点を獲得している．さらに，ファイルの順処理の間はどの節も2回以上アクセスされないから，主記憶ではわずかに節1個分の場所が利用できさえすればよい．このように B^+ 木は，ランダム処理と順処理の両方を伴う応用に大変適している．

接頭辞 B^+ 木 (Prefix B^+-Trees)

B^+ 木における索引と シーケンスセットの 分離には，何か直観に訴えるものがある．索引部には，探索を正しい葉に導くための道路地図としての役割しかなかったことを想起されたい．とすると，そこには実際のキーを入れておく必要はまったくないのである．キーが文字列から成っているとき，実際のキーを分離値として使わないことには，十分な理由がある．実際のキーは場所を食いすぎるのである．Bayer と Unterauer[BAYE 77] が，接頭辞 B^+ 木なる代案を考えている†．

"binary", "compiler", "computer", "electronic", "program", "system" という一連の英字キーが，図 15 のように B^+ 木に割り付けられていると仮定しよう．索引部

† （訳注） 以下に説明されているものは，BAYE 77 において「単純接頭辞B木」と呼ばれているものである．Bayer らが「接頭辞B木」と呼んでいるものは，これをさらに発展させたものである（p. 33 †1 参照）．

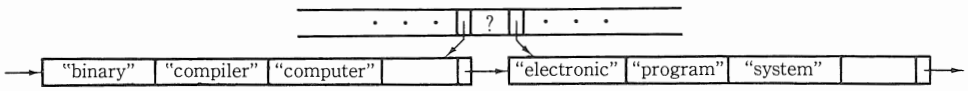


図 15　接頭辞 B⁺木の一部．“computer” と “electronic” を分離する索引項目は “e” で十分である

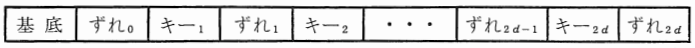


図 16　圧縮ポインタをもった節．i 番目のポインタを得るには，i 番目の「ずれの値」に基底を加算する

における “computer” と “electronic” の間の分離値はこのいずれかである必要はなく，これらの間にあるものなら何でもよい．たとえば “elec”, “e”, “d” のどれでもうまくゆくのである．検索処理に何ら変わるところはないのだから，場所の節約のためにはそのような分離値のうち最短のものを用いるべきである．必要な空間が小さくなると，より多くのキーが個々の節に置け，分岐係数が増し，木の高さが減る．低い木のほうが検索に費用がかからないから，短い分離値を用いると場所の節約になるだけでなくアクセス時間の節約にもなる．

分離値として使うための，キーの最短の一意的な接頭辞を選ぶには，簡単な手法でうまくゆく．さきの例では，“electronic” と “computer” を区別する最短の接頭辞は “e” である．しかしときたま，接頭辞の技法ではうまくゆかないこともある．“programmers” を “programmer” と区別する最短の接頭辞を選んでも，何の節約にもならない．Bayer と Unterauer は，そのような場合には，分離の算法に都合のよい対を得るためにキーの近傍を調べてまわることを提案している．こうすると節にキーが不均等に載ったままになるかもしれないが，節の一方に，二，三の余分なキーがあったとしても全体の費用には影響しないであろう．

仮想B木 (Virtual B-Trees)

現代の計算機システムの多くは，個々のユーザに大きな仮想記憶を与える記憶管理方式を採用している．ユーザの仮想記憶のアドレス空間はページに分けられて二次記憶に退避してあり，参照されたとき自動的に主記憶に載せられる．この技法は**デマンドページング**と呼ばれ，ユーザ間での実記憶の多重利用を実現すると同時に，あるユーザが他人のデータに手を出さないことを保証するための保護も行なってくれる．さらに，二次記憶との間のデータ転送を高速で行なうために，専用のハードウェアがページングを取り扱う．

デマンドページングのハードウェアが使えるということは，B木の興味深い実現方式を示唆している．注意深い割付けを行なえば，B木の個々の節を仮想記憶空間の1ページに対応させることができる．するとユーザは，B木をあたかも，それが主記憶上にあるかのように扱えばよくなる．主記憶上にない節（ページ）へのアクセスは，システムによる二次記憶からの節の「ページ・イン」をひきおこすのである．

たいていのページング算法は，新しいページのための場所をつくるのに，最長未使用 (least recently used, LRU) のページを追い出すことにしている．B木の用語でいえば，最もよく使われる節は根に近い節であり，それらは主記憶内にとどまる傾向がある．実は，Bayer と McCreight [BAYE 72], Knuth [KNUT 73] の両者が，ページングのハードウェアが使えないときでも LRU 機構を用いることを提唱している．少なくとも根は，毎回の探索でアクセスされるのだから主記憶に残っているべきである．

そういうわけで，仮想B木の利点をまとめると次のようになる．

1) 専用のハードウェアが高速でデータ転送を行なう．
2) 記憶保護機構が他のユーザを隔離する†1.
3) 木のうちのよくアクセスされる部分が，主記憶にとどまる傾向をもつ．

圧縮 (Compression)

B木の性能を改善するために，ほかにもいくつかの技法が提案されている．Wagner [WAGN 73] がそのいくつかを要約しており，そのなかに圧縮キーと圧縮ポインタの概念が含まれている†2.

ポインタは，絶対番地でなく基底／変位 (base/dis-

†1 (訳注)　これを仮想B木の利点として挙げることにはやや疑問もある．なぜなら，これは仮想記憶方式の一般的な特徴であって，B木の記憶管理を仮想記憶機構に委ねることで初めて得られる御利益ではないように思えるからである．実際，仮想B木でないB木を仮想記憶方式の計算機で扱っても，主記憶のデータは同様に保護されるし，二次記憶上のデータも同程度に安全であるといえよう．

†2 AUER 76 も参照されたい．

(原注)

placement) の形式を用いることにより圧縮される．圧縮ポインタは図 16 に示されるような形をしており，基底番地は節にひとつだけ格納され，あとはずれ (offset) の値，すなわち基底からの変位が個々のポインタの代わりに用いられる．実際のポインタ値を再構成するには，そのポインタの変位に基底の値を加算すればよい．圧縮ポインタの技法は，ポインタが大きなアドレス値をとる仮想B木にとくに適している．

キー，あるいは分離値は，冗長さをなくすためのいくつかの標準的な技法 [RUBI 76] のうちのいずれかを用いて圧縮することができる．キーの圧縮とポインタの圧縮はどちらも個々の節の容量を大きくし，それによって検索費用を低減させる．ただ二次記憶へのアクセス回数の減少とひきかえに，いったん主記憶に読み込まれたあと，節のなかを探索するのに必要な CPU 時間は増加する．したがって，こみいった圧縮技法がいつも費用の面で有効だとは限らない．

キーに対しては，前・後部圧縮 (front and rear compression)†1 の双方が適用できるということを書き留めておかねばなるまい．たとえば，Bayer と Unterauer [BAYE 77] が接頭辞 B^+ 木のためのキーの圧縮について考察している．

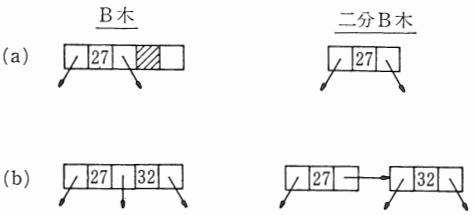


図 17 B木の節と，それに対応する二分B木の節．二分B木表現における右ポインタは，兄弟または子を指すことができる

可変長項目

多くの応用において，可変長のキーをもったデータの記憶が必要とされる．これに加えて，可変長の項目は上述の圧縮技法を用いた結果としても生じる．McCreight [MCCR 77] が，可変長の項目をもった木の記憶について考察しており，挿入の際により短いキーを昇進させると，いかに記憶効率が良く，アクセス時間の短いB木になるかを示している．

二分B木

Bayer [BAYE 72a] によって提案されたもうひとつの変形，つまり二分B木は，B木を単一レベル記憶†2 に適したものにする．本質的には二分B木は位数1のB木であり，個々の節は1個ないし2個のキーと2個ないし3個のポインタをもつ．ただ半分しか埋まっていない節のために場所を浪費するのを防ぐため，図 17 に示されるようなつなぎによる表現が用いられるのである．1個のキーをもつ節がまったく図 17(a)†3 にあるとおりに表現されるのに対し，2個のキーをもつ節は図 17(b) にあるようにポインタでつながれている．節の右側のポインタは兄弟と子のいずれかを指すので，その意味を示すために余分な1ビットを用いなければならない．

解析を行なってみると，挿入・削除・検索にはB木の場合と同様，やはり $\log n$ ステップしかかからないが，最も右側の路を探索すると，最も左側の路を探索したときの2倍の節をアクセスしなければならないことが示される．また右側のポインタをふたつの目的に用いることが，挿入と削除の算法を複雑化する．対数的な費用を維持するためには，ふたつの右ポインタが連続して兄弟の節を指すことのないよう注意しなければならない．2本以上†4 連続した兄弟へのつなぎを防ぐ，節の回転処理†5

---

†1 (訳注) 両者を，B木への応用に則して簡単に説明する．まず後部圧縮とは，たとえば programmers と programming の分離キーとして，最短の非共通接頭辞，つまり programme あるいは programmi を用いようというものである．接頭辞 B^+木(BAYE 77 の単純接頭辞B木) に使われたのと同じ考え方である．一方，前部圧縮とは次のようなものである．

文字列をキーとするB木の探索では，キーの同定は一挙にではなく徐々に進む (たとえば，3段下りたところでは pro まで，4段おりたところでは program まで，というように) と考えられる．したがって個々の節は，「そこにくるまでに同定されているはずの接頭辞」をもっているわけであり，その接頭辞の部分は比較対象からはずしてしまっても何らさしつかえない．たとえば program までが同定されている場合は，最初の例の分離キーは mers か ming だけで十分なのである．

このふたつの技法を組み合わせると，分離キーはさらに me または mi にまで圧縮できる．これを前・後部圧縮というのである．実は，BAYE 77 における「接頭辞B木」と「単純接頭辞B木」の違いは，前部圧縮まで行なうかどうかの違いなのである．

†2 (訳注) 単一レベル記憶 (one-level store) とは，ファイル用の二次記憶まで，主記憶のレベルに統合化してしまい，すべてのデータをひとつのアドレス空間のなかで管理する方式である [武井 80]．

†3 (訳注) 原論文では図17(a)と図17(b)が入れ換わっている．

†4 (訳注) 原文では three or more となっている．

†5 (訳注) 回転処理とは，ポインタのつけかえ，および/または意味の変更 (親子関係↔兄弟関係) による，節の位置関係の局所的・組織的修正のことを指す．

の詳細な算法は，BAYE 72a と WIRT 76 に与えられている．

左右両方のつなぎが兄弟を指すことを考慮した二分B木の拡張版は，二分B木にはない対称性を示す．そこでそのようなデータ構造に対し，Bayer［BAYE 73］によって対称二分B木という名がつけられた．彼はまた，対称二分B木がその副クラスとして，有名な AVL 木［FOST 65］のクラスを含んでいることを報告している．

2-3 木と理論上の成果

Hopcroft は 2-3 木(2-3 tree) の概念を生み出し†1，その単一レベル記憶における有用性を探った．2-3 木における個々の節は2ないし3個の子をもつ（というのは1ないし2個のキーをもつからである）．したがって 2-3 木は位数1のB木であり，その逆も真である．2-3 木は節が小さいため外部記憶用としては非実用的であるが，内部データ構造にはまことに適している．Rosenbaum と Snyder［ROSE 78］，そして Miller ら［MILL 77］は，与えられたキーの組に対して最適な 2-3 木を構成する問題を考察している．彼らは費用の基準として，比較回数と節へのアクセス回数をそれぞれ用いている．おのおのの場合について，整列されたキーの並びから最適な木を構成する線形時間の算法が与えられている．MILL 77 の結果は，任意の位数のB木にも拡張される．

Yao［YAO 78］は，一様分布をなす n 個のキーの組からつくられた 2-3 木の解析結果を報告している．この論文は記憶効率の期待値の上限と下限の両方を与えている．Yao はさらに解析を高位のB木にまで拡張し，記憶効率の期待値が $\ln 2 \approx 69\%$ であることを示している．

Guibas ら［GUIB 77］は，アクセスの確率に大きな偏りがあるようなキーの列を保守するための，B木の変形を考えている．関心がもたれている場所を指す一組の「指」(finger) を維持することにより，指から p 個以内のところにある項目が $\log_d p$ 時間で更新できるというのである．たとえば，指をキーの並びの最初と最後に位置づけておいたとしよう．活動の盛んな場所が移るにつれ，指のうちの一本を新しい場所へと動かすことができるのである．

Guibas と Sadgewick［GUIB 78］は，また別のB木の方式を与え，いくつか均衡木の技法の効率を比較している．彼らの重要な投稿には，上向きの分割処理がけっして必要でないことが示されている．その秘訣は，木を下っているときに，ほとんど一杯になっている節を割ってしまうことにある．次節で，下からの更新を排除することが効率に決定的にきいてくることを示す．

なお，関連する理論上の成果については，BROW 78 と BROW 78a も参照されたい．

4. 複数ユーザ環境におけるB木

もしB木を汎用データベース・システムで使おうとするならば，何人かのユーザの要求が同時に処理できるようになっていなければならない．その際プロセスの同期をとるために何か制約をつけないと，プロセスどうしが互いに干渉しあうかもしれない．あるプロセスが節に変更を加えているとき，別のプロセスがそれを読み取って，そのなかのポインタをたどろうとするかもしれない．さらに相互作用がこみいってくると，挿入や削除の操作が下から上へのアクセスを要しているときに，検索操作が上から下へと処理を開始したりする．Samadi［SAMA 76］は，並行動作の問題にひとつの解答を与えている．Held と Stonebraker［HELD 78］は，並行処理における競合は，ひとつのプロセスしか木にアクセスさせないことにすれば解決するけれども，複数ユーザ環境におけるB木の優位性を損わせている，と論じている．

Bayer と Schkolnick［BAYE 77a］は，管理プロセスによって施行される一組の施錠規約 (locking protocol) によって，並行動作を許しながらB木の正しい状態を保証することができることを示している．本質的には，まず検索のプロセスはいったん読み込んだ節の施錠，あるいは差押え (hold) を行なって，他のプロセスがその節に手出ししないようにする．探索が次のレベルへ進むと，検索プロセスはその先祖にかけていた錠を解き，それを他のプロセスが読み取ることを許すのである．このように，読み手はいつでも高々2個の節しか施錠せず，他の読込みプロセスは木の他の部分を同時に探索（し，施錠）することができるのである†2．

次に，並行処理の環境における更新は，もっと複雑な規約を要するようなもっと複雑な問題を提起する．更新は，木の高いレベルのほうに影響を及ぼすかもしれない

†1 (訳注) KNUT 73 によると，これに関する原論文は "unpublished" だそうである．

†2 (訳注) 原文ではこのように，同じ節を複数の読み手が同時にアクセスしてはいけないような書き方がなされているが，読み手をひとつに制限する理由は別にないし，現に BAYE 77a でもこのような施錠規約を用いてはいない．通常は読み手用の共有錠 (share lock) (あるいは読出し錠 (read lock)) と，書き手用の専有錠 (exclusive lock) (あるいは絶対錠 (absolute lock)) とが用意され，共有錠どうしに限って共存を認めるという方式がとられる．

から，更新プロセスは自分がアクセスした節を予約状態にしておき，その節を施錠する権利を留保する．あとで更新プロセスが，自分の行なう変更が予約された節に伝播すると判断すると，その節の予約が錠に変更される．反対に，更新が予約状態の節に影響しないということになれば，その節の予約が取り消される．読み手は必ず葉まで行ってしまうので，予約された節を読むことはかまわない．しかし最初の予約が取り消されるまでは，二度目の予約を受けることはできない．

ひとたび更新プロセスが，ある葉にいたる節を予約しおえると，今度はその予約を上から順に，絶対錠 (absolute lock) に変換してゆく．絶対錠は，他のプロセスがその節へアクセスしないことを保証する．それから更新が進行し，絶対錠のかかった節だけが変更されてゆく．すべての変更がすむと絶対錠がとりはずされ，更新された路は他のプロセスにも使えるようになる．

根から葉にいたる路全体を予約すると，他の更新プロセスはB木にアクセスできなくなる．しかも，大多数の更新処理はほんの数レベル——葉に近いほうの——にしか影響を及ぼさない．したがって路全体を予約することは望ましいことではない†1．しかし予約を少なくしすぎると，根からやり直しになってしまうかもしれない．Bayer と Schkolnick は，これらの両極端の間のトレードオフを表わした，一般化された施錠規約を提案している．彼らはパラメタ化されたモデルを与え，いかに予約方式が，現在の技術を活かすに十分な並列性を許し，同時に再予約のためにほとんど時間を浪費せずに済んでいるかを示している．

これに対して，GUIB 78 に提案された上からの分割法を用いると，更新プロセスが木をさかのぼる必要がなくなるので，最も単純な規約以外の規約が一切不要になる．そしてある与えられた時点で見ると，たった一組の節しか施錠されずに済んでいるであろう．ただもちろん，節を一杯になる前に割ってしまうことの代償として，記憶効率がわずかに低下し，それに対応してアクセス時間が増加するということはある．

機密保護 (Security)

複数ユーザ環境における情報の保護は，データベース設計者にもうひとつの問題を提起する．さきに，仮想B木の話題のところで，ページングのもつ記憶保護機能からユーザ間の分離が達成できることが示された．しかしファイルの中身がシステムの外で保護されなければならないときは，何らかの暗号化技法を用いなければならない．Bayer と Metzer [BAYE 76] は暗号化の技法と，起こりそうな安全への脅威について考察している．彼らは，ハードウェアで実現しない限り，暗号化には比較的高い費用がかかることを明らかにしている．しかし一方，暗号化されたファイルを取り扱うためにB木の保守の算法に加えなければならない変更は，とくにデータ転送のときにいっしょに暗号化をやってしまえるならば，少なくて済む．

5. B⁺ 木を用いた汎用アクセス法

この章では B⁺ 木の使用例を紹介する．それは，IBM のB木をもとにした汎用アクセス法である，VSAM [IBM 1, IBM 2, KEEH 74, WAGN 73] のことである．VSAM は，広汎な応用分野で使うことを意図して，対数的費用の挿入・削除・検索操作のほかに順探索をも支援している．伝統的な索引順編成と比較して，B⁺ 木は次のような利点を提供している†2．それは記憶域の動的な割当てと解放，50%以上と保証された記憶効率，そしてファイル全体の定期的「再編成」が不要なことである．

VSAM はキーと関連情報の両方の記憶域を操らなければならないから，VSAM ファイルは図 18 に示すように表現されている．VSAM の木のうち上のふたつの部分は，さきに述べた B⁺ 木の索引部とシーケンスセットを形成しており，葉が実際の**データ・レコード**を収容している．VSAM の用語では葉は**コントロール・インターバル**と呼ばれ，1回の入出力操作で転送されるデータの基本単位をなしている．それぞれのコントロール・インターバルは1個またはそれ以上のデータ・レコードを，そのインターバルの形式を記述した**制御情報**とともに収容している．図 19 は，コントロール・インターバルの各欄を示すものである．

†1 (訳注) このあたりの記述からは，更新プロセスが，葉にたどりつくまでは上位のレベルの予約を取り消さないようにも読みとれるが，実はそうする必要はない．予約をしながら木を下ってくる途中に，分割寸前でない節（削除処理では，連結寸前でない節）があった場合は，その上位にまで分割や連結が伝播することはないから，その節より上位の予約をただちに取り消してしまってかまわない．BAYE 77 a の算法もそうなっている．

†2 (訳注) これは B⁺ 木の利点であって，必ずしも VSAM の利点ではないことに注意されたい．実際，VSAM では節が連結されたり解放されたりすることはなく，したがって記憶効率が50%を下回ることもありうる．

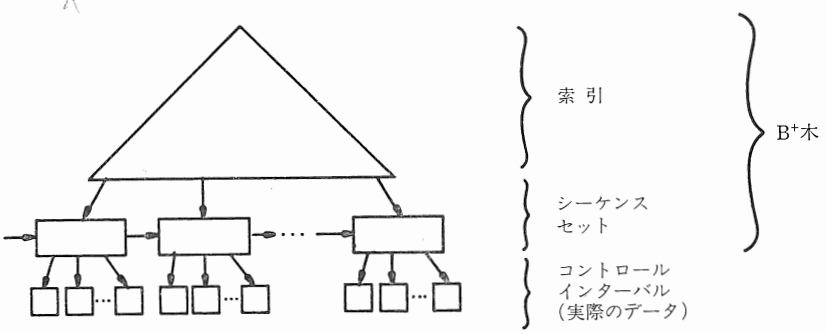


図 18　実際のデータ（関連情報）を葉に蓄えた VSAM ファイル

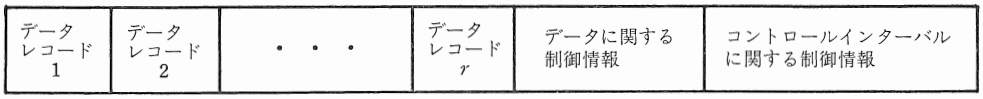


図 19　コントロール・インターバルの形式．制御欄は，コントロール・インターバル自身，およびデータ欄の形式について記述している

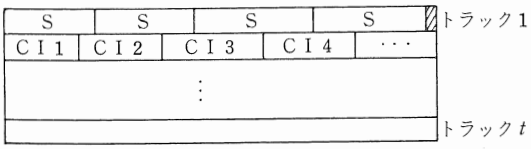


図 20　待ち時間を最小にするために，シーケンスセットの節 S を繰返し記録してあるコントロール・エリアの形式

性能の向上

VSAM がデータに対する論理的で機械独立な見方をユーザに提供しているとはいっても，処理を効率的に行なおうとするならば，ファイル編成を，それが載る装置に適合させなければならない．したがって，コントロール・インターバルの最大長は，ハードウェアが1回の操作で転送できるデータの最大の構成単位によって制限される．それに加えて，ひとつのシーケンスセットの節に関連した全コントロール・インターバル（**コントロール・エリア**と呼ばれる）が，そのファイルを格納するのに使われている，特定のディスク記憶の1シリンダに入らなければならない．これらの制約は効率を改善し，さらに以下で述べるような性能向上さえも可能にするのである．

ひとつのシーケンスセットの節のすべての子が1シリンダに割り付けられているので，さらに同じシリンダにシーケンスセットの節も割り付けてしまうと，効率が改善される．そうすると，いったんシーケンスセットの節を読み出したあとで，ディスクのアームを動かさずにコントロール・エリアのなかの項目を取り出すことができるのである．図 20 は，シーケンスセットの節をデータのそばに割り付ける方法をさらに拡張したもので，シリンダのなかの1トラックに，シーケンスセットの節が繰り返し記録されている様子を示している．写しをもっていると，ディスクの回転待ち時間を減らすことができるのである．VSAM はほかにもいくつかの方法で性能の改善を試みている．ポインタはさきに述べた基底/変位の方法で圧縮され，キーは前（接頭辞）と後（接尾辞）の両方向に圧縮され，索引レコードはいくつも複写しておくことができ，索引とデータへの同時アクセスを可能にするために索引を別の装置に割り付けることができる．さらに，VSAM では索引部を仮想B木とし，検索に仮想記憶のハードウェアを用いるようにすることができるのである．

木構造化されたファイル登録簿

おそらく VSAM の実現方式のなかに見られる最も新鮮な考え方は，システム全体を通じて単一のデータ形式を用いるべきだという考え方であろう．たとえば，システム内のすべての VSAM ファイルの登録簿(directory)を保守するプログラムは，**マスタ・カタログ**と呼ばれる VSAM ファイルに情報を蓄えている．図 21 は，個々の VSAM ファイル（あるいは VSAM データセット）に関する登録事項をもったマスタ・カタログを示すものである．すべての VSAM ファイルがこのカタログに登録されなければならないから，ファイル名さえ与えればシステムはどのファイルでも自動的に見つけ出してくる．もちろん，カタログもまた VSAM データセットなので，カタログは自分自身について記述した項目ももっている．

いくつかのプロセスがマスタ・カタログにアクセスし

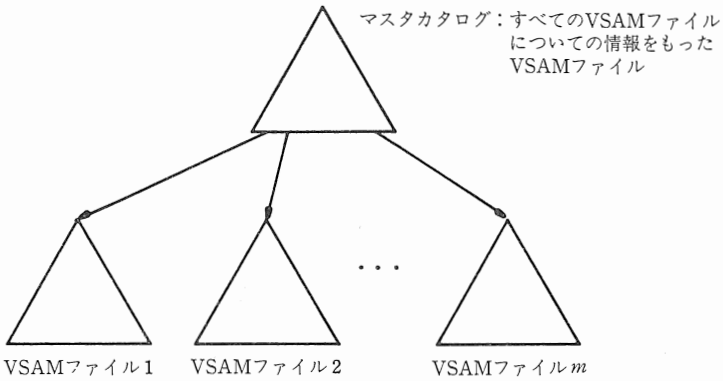


図 21　すべての VSAM ファイルの登録簿として使われる VSAM
マスタ・カタログは，それ自身 VSAM ファイルである

ようとすると，競合が起きて，ひとつを除くすべてのプロセスが待たねばならなくなる．このような競合によって引き起こされる長々しい遅延を防ぐために，各ユーザは自分の VSAM ファイルを登録した局所的なカタログを定義することができる．このとき，VSAM ファイルであるユーザ・カタログは，マスタ・カタログに登録されなければならない．マスタ・カタログの探索によっていったんユーザ・カタログが見つかってしまうと，あとの，ユーザ・カタログで索引づけられたファイルの参照は，マスタ・カタログの探索を伴わずにできる．このようにしてできた，多レベルの木構造化されたカタログ方式には，MULTICS のファイル・システム[ORGA 72] に似た趣がある．

他の VSAM 機能

われわれの短い議論では，VSAM の多くの面が明らかになっていない．読者は，われわれが VSAM をざっと概観したにすぎないということに注意していただきたい．一例をあげると，われわれの議論した VSAM ファイルは**キー順** (key-sequenced) ファイルと呼ばれる．レコードにキーがついていないとき（すなわち，キーを用いた操作が行なわれないとき)は，もうひとつの形である**エントリ順** (entry-sequenced) の VSAM ファイルを使えば効率的な順処理が行なえる．エントリ順の VSAM ファイルは索引を必要としないから，保守にかかる費用が少ない．

またシステムは，VSAM ファイルの保守と検索の手続きに加え，VSAM ファイルを定義し，初期作成を行なう機能も提供している．ユーザはファイル内の空き領域をどのように分散させるかを決定しなければならない．もしユーザが多数の挿入を見越しているならば，おそらく，個々の節が 100% 埋まるような初期作成をすべきではなかろう．そうしてしまうと，最初の挿入が高くつくからである．一方，ファイルが比較的安定な状態を保つならば，容量の50%しか節に詰めないのは記憶域の浪費である．VSAM ファイル定義機能は，選ばれたパラメタに従って，ファイルの初期作成を行なう手助けをしてくれる．

最後に，VSAM は大量の連続したレコードの組を効率的に挿入する機能，データ保護機能，ファイル・バックアップ機能，障害回復機能を供給していることをつけ加えておく．これらはみな，生産的環境において必要となるものである．

まとめ

均衡多分木構造の外部ファイル編成である B 木は，効率的で，多才で，単純で，しかも保守が容易である．そのひとつの変形である B^+ 木は，検索・挿入・削除に対して対数的という望ましい費用特性を保ちつつ，効率的な順処理を可能にしている．B 木の方式は，ファイルがふくらんだり縮んだりするにつれて記憶域を割り当てたり解放したりして，常に50%の記憶効率を保証する．さらに，B 木は厳密に正反対の方法でふくらんだり縮んだりするから，処理のためのやりとりを激しく行なった後でも，手間のかかるファイルの「再編成」はけっして必要がない．

B 木の異なった実現手法は，より良い性能や一般性をもたらしたり，B 木を複数ユーザ環境で用いることを可能にしたりする．キーやポインタの圧縮，二次記憶上への節の慎重な割付け（および多重記録），そして挿入・削除時の局所的なキーの再配分は，いずれも性能を向上させ，B 木を生産的環境にも耐えるものにする．さらに施

錠規約や，仮想記憶の保護機能や，データの暗号化は，B木を何人かのユーザで共有する際に必要となる機密保護と相互排除とを行なってくれる．

IBM の VSAM は，B木をもとに汎用アクセス法を構成することが適当であることを実証している．ユーザのファイルがB木であることに加え，システム自身もすべての VSAM ファイルの名前とありかを登録するためにB木ファイルを用いている．VSAM は，効率的な順処理を可能にするために B^+ 木の実現方式を利用し，さらに性能向上とデータ保護のために，利用可能な技法の多くを組み入れている．

謝　辞

筆者は，査読者の方々に謝意を表します．とくに，査読者の方々からB木の歴史についてご連絡をいただいたことに対してお礼を申し上げます．また，他の競合メーカがアクセス法の詳細を公開しないなかで，B木に基づくアクセス法に関する詳細な情報を快く提供してくださった IBM 社にも感謝いたします．

参考文献

AHO74　AHO, A., HOPCROFT, J., AND ULLMAN, J. *The design and analysis of computer algorithms,* Addison Wesley, Publ. Co., Reading, Mass., 1974.

AUER76　AUER, R. *Schlusselkompressionen in B*-baumen*, Diplomarbeit, Tech. Universitat, Munich, 1976.

BAYE72　BAYER, R., AND MCCREIGHT, C. "Organization and maintenance of large ordered indexes," *Acta Inf.* **1,** 3 (1972), 173–189.

BAYE72a　BAYER, R. "Binary B-trees for virtual memory," in *Proc. 1971 ACM SIGFIDET Workshop*, ACM, New York, 219–235.

BAYE73　BAYER, R. "Symmetric binary B-trees: data structure and maintenance algorithms," *Acta Inf.* **1,** 4 (1972), 290–306.

BAYE76　BAYER, R., AND METZGER, J. "On encipherment of search trees and random access files," *ACM Trans. Database Syst.* **1,** 1 (March 1976), 37–52.

BAYE77　BAYER, R., AND UNTERAUER, K. "Prefix B-trees," *ACM Trans. Database Syst.* **2,** 1 (March 1977), 11–26.

BAYE77a　BAYER, R., AND SCHKOLNICK, M. "Concurrency of operations on B-trees," *Acta Inf.* **9,** 1 (1977), 1–21.

BERL78　BERLINER, H. *The B*-tree search algorithm: a best-first proof procedure,* Tech. Rep. CMU-CA-78-112, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, 1978.

BROW78　BROWN, M. "A storage scheme for height-balanced trees," *Inf. Process. Lett.* **7,** 5 (Aug. 1978), 231–232.

BROW78a　BROWN, M. "A partial analysis of height-balanced trees," *SIAM J. Comput.*, to appear.

CLAM64　CLAMPETT, H. "Randomized binary searching with tree structures," *Commun. ACM* **7,** 3 (March 1964), 163–165.

FOST65　FOSTER, C. "Information storage and retrieval using AVL trees," in *Proc. ACM 20th National Conf.*, ACM, New York, 1965, 192–205.

GHOS69　GHOSH, S., AND SENKO, M. "File organization: on the selection of random access index points for sequential files," *J. ACM* **16,** 4 (Oct. 1969), 569–579.

GUIB77　GUIBUS, L., MCCREIGHT, E., PLASS, M., AND ROBERTS, J. "A new representation for linear lists," in *Proc. 9th ACM Symp. Theory of Computing*, ACM, New York, 1977, 49–60.

GUIB78　GUIBAS, L., AND SEDGEWICK, R. "A dichromatic framework for balanced trees," in *Proc. 19th Symp. Foundations of Computer Science*, 1978, 8–21.

HELD78　HELD, G., AND STONEBRAKER, M. "B-trees reexamined," *Commun. ACM* **21,** 2 (Feb. 1978), 139–143.

HORO76　HOROWITZ, E., AND SAHNI, S. *Fundamentals of data structures*, Computer Science Press, Inc., Woodland Hills, Calif., 1976.

IBM1　*OS/VS Virtual Storage Access Method (VSAM) planning guide*, Order No. GC26-3799, IBM, Armonk, N.Y.

IBM2　*OS/VS Virtual Storage Access Method (VSAM) logic*, Order No. SY26-3841, IBM, Armonk, N.Y.

KARL76　KARLTON, P., FULLER, S., SCROGGS, R., AND KACHLER, E. "Performance of height balanced trees," *Commun. ACM* **19,** 1 (Jan. 1976), 23–28.

KEEH74　KEEHN, D., AND LACY, J. "VSAM data set design parameters," *IBM Syst. J.* **3,** (1974), 186–212.

KNUT68　KNUTH, D. *The art of computer programming, Vol. 1: fundamental algorithms*, Addison-Wesley Publ. Co., Reading, Mass., 1968.

KNUT73　KNUTH, D. *The art of computer programming, Vol. 3: sorting and searching*, Addison-Wesley Publ. Co., Reading, Mass., 1973.

MARU77　MARUYAMA, K., AND SMITH, S. "Analysis of design alternatives for virtual memory indexes," *Commun. ACM* **20,** 4 (April 1977), 245–254.

MAUE75　MAUER, W., AND LEWIS, T. "Hash table methods," *Comput. Surv.* **7,** 1 (March 1975), 5–19.

MCCR77　MCCREIGHT, E. "Pagination of B*-trees with variable-length records," *Commun. ACM* **20,** 9 (Sept. 1977), 670–674.

MILL77　MILLER, R., PIPPENGER, N., ROSENBERG, A., AND SNYDER, L. *Optimal 2-3 trees*, IBM Research Rep. RC 6505, IBM Research Lab., Yorktown Heights, N.Y., 1977.

NIEV74　NIEVERGELT, J. "Binary search trees and file organization," *Comput. Surv.* **6,** 3 (Sept. 1973), 195–207.

ORGA72　ORGANICK, E. *The Multics system: an examination of its structure*, MIT Press, Cambridge, Mass., 1972.

ROSE78　ROSENBERG, A., AND SNYDER, L. "Minimal comparison 2-3 trees," SIAM *J. Comput.* **7,** 4 (Nov. 1978), 465–480.

RUBI76　RUBIN, F. "Experiments in text file compression," *Commun. ACM* **19,** 11 (Nov. 1976), 617–623.

SAMA76　SAMADI, B. "B-trees in a system with multiple views," *Inf. Process. Lett.* **5,** 4

(Oct. 1976), 107–112.

SEVE74 SEVERENCE, D. "Identifier search mechnisms: a survey and generalized model," *Comput. Surv.* **6**, 3 (Sept. 1974), 175–194.

WAGN73 WAGNER, R. "Indexing design considerations," *IBM Syst. J.* **4**, (1973), 351–367.

WEDE74 WEDEKIND, H. "On the selection of access paths in a database system," in *Data base management (Proc. IFIP Working Conf. Data Base Management)* J. Klimbie and K. Koffeman (Eds.), Elsevier/North-Holland Publishing Co., New York, 1974, 385–397.

WIRT76 WIRTH, N. *Algorithms + data structures = programs*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

YAO78 YAO, A. "On random 2-3 trees," *Acta Inf.* **9**, 2 (1978), 159–170.

訳者付記

(KNUT 68 の邦訳 (二分冊))

広瀬 健訳：「基本算法 / 基礎概念」，サイエンス社 (1978).

米田信夫・筧 捷彦訳：「基本算法 / 情報構造」，サイエンス社 (1978).

(ORGA 72 の邦訳 (二分冊))

菊池豊彦・佐々木彬夫訳：「MULTICS システム(上・下)」，共立出版 (上 1973，下 1974).

(WIRT 76 の邦訳)

片山卓也訳：「アルゴリズム＋データ構造＝プログラム」，日本コンピュータ協会 (1979).

SCHN 78 Schneiderman, B. "Jump searching : a fast sequential search technique", Commun. ACM **21**, 10 (Oct. 1978), 831–834.

PERL 78 Perl, L., Itai, A., and Avni, H. "Interpolation search—A log log N search", Commun. ACM **21**, 7 (July 1978), 550–553.

植村 79 植村俊亮：「データベースシステムの基礎」，オーム社 (1979).

武井 80 武井欣二：「仮想記憶方式」，情報処理 **21**, 4 (1980年4月号), 358–368.

図 7～図 11 は訳者が書き直したものです.

付 録

[この論文に関して読者から Comer へ寄せられた手紙の要約と，それらに対する謝礼が，第 11 巻第 4 号の Surveyor's Forum p. 412 に掲載された．以下はその訳である.]

Comer からの手紙

「広くゆきわたったB木」が世に出て以来，何人かの読者の方から，この概説に触れられていない成果についてのご教示の手紙をいただきました．私はそのすべてに感謝の意を表するとともに，いただいた情報を下の参考文献表の形でご覧にいれます．またこれらの調査報告に加え，Honeywell 社からは，B木を用いたふたつの異なる同社のファイル・システム製品について，ご説明をいただきました．そのひとつは GCOS ファイル・システムで，IBM の VSAM と同様の応用です．もうひとつは vfile モジュールといって，Multics 操作システムの一部をなすものです.

Sperry Univac 社の Richard Wexelblat 氏からは，1960 年代の初期に Pensylvania 大学の Multi-List Project において，H. J. Gray 教授と Noah Prywes 教授の指導の下で行なわれた，均衡木構造についての形成的な研究についてご教示をいただきました．彼によれば，それに最も関連した文献は下に掲げてある Waltzer Landauer 氏の論文だそうです.

McMaster 大学の D. Wood 教授は，どこを見回してもまたB木の論文が見つかる，というほどに，B木は本当に広くゆきわたっていると述べてくださいました．B木に関する研究の分野はこれほど活発なので，これですべての関連文献を把握したと考えるのは愚かなことです．そこで，私が見逃してしまった業績をあげられた方々に対し，前もってお詫びを申し上げておくことにします.

Douglas Comer
Computer Science Department
Purdue Univesity
West Lafeyette, Ind. 47907

BERK72 BERKOVICH, S. "Machine organization of a growing search tree," *Dolk. Akad. Nauk SSSR* **202**, 2 (1972).

CULI79 CULIK, K., OTTMAN, T., AND WOOD, D. *Dense multiway trees*, Tech. Rep. 79-CS-5, McMaster U., Hamilton, Ont., Canada, 315–344.

FAGI79 FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. "Extendible hashing—A fast access method for dynamic files," *ACM Trans. Database Syst.* **4**, 3 (Sept. 1979), 315–344.

KWON78 KWONG, Y. S., AND WOOD, D. *T-trees: A variant of B-trees*, Tech. Rep. 78-CS-18, McMaster U., Hamilton, Ont., Canada, 1978.

LAND63 LANDAUER, W. "The balanced tree and its utilization in information retrieval." *IEEE Trans. Comput.* (Dec. 1963).

LITW78 LITWIN, W. "Virtual hashing: A dynamically changing hashing," in *Proc. 1978 Very Large Data Base Conf.* (Berlin), pp. 517–523,

LOME78 LOMET, D. *Multi-table search for B-tree files*, Tech. Rep. RC 7461, IBM T. J. Watson Research Ctr., Yorktown Heights, N.Y., 1978.

ROSE78 ROSENBERG, A. AND SNYDER, L. *Compact B-trees, Tech. Rep.* RC 7343, IBM T. J. Watson Research Ctr., Yorktown Heights, N.Y., 1978.

ZAKI77 ZAKI, A. *Insertion and search in 2-3 trees vs HB(1) trees,* Ph.D. Diss., U. of Washington, Seattle, Wash., 1977.