B+木に関する調査

慶應義塾大学
環境情報学部
2023年3月
平田　朋

B+木に関する調査

学籍番号：71846936
氏名：平田　朋
Tomo Hirata

近年の情報社会では、データの管理・保存がかつてないほど重視されており、急速な経済発展に伴い、日々確保すべきデータ量が爆発的に増加しているため、効率的かつ確実な管理方法が求められている。B+木は、B 木の一般的な変種として広く知られており、バルクデータに最も適したツリー型データ構造であると考えられている。本論文は、その B+木の基本的なこと、動作、B+木の変種、関連研究についてて調査したものをまとめた。私は、今回、調査した B+木の更なる研究により、データ処理時間を短縮しながら効率的なインデックス構造化を提供することが、今後の情報社会において重要だと考える。

研究指導教員：川島 英之

目 次

図 目 次

第1章　はじめに

1.1　研究背景

2022 年 6 月 DATAVERSITY が発行したレポートによると、世界中で生成されるデータの量が指数関数的に増加するにつれて、データ駆動型の企業は、そのすべてのデータを保存、管理、および処理するために、より革新的なデータベース管理システムにますます注目していると述べている [1]。B+木は、B-木の一般的なバリエーションとして広く知られており、バルクデータに最も適したツリー型のデータ構造と見なされている [2]。データの収集、保存、分析の重要性が日々高まる中、企業は、データベース管理システムの改善とデータ管理業務の遂行とより生産的な成果の達成の貢献を目指している。このようなデータベース管理システムの改善の実現に向けて、B+木が積極的に活用されていくとされている。

B+木の派生元なっているデータ構造に B-木があるが、computer science において、B-木はソートされたデータを保持し、対数時間で検索、順次アクセス、挿入、削除を可能にする自己平衡木データ構造である。B-木をは二分探索木を一般化したもので、2 つ以上の子を持つノードを許容する。他の自己平衡二分探索期とは異なり、B-木は、データベースやファイルシステムなど、比較的大きなデータブロックを読み書きするストレージシステムに適している [4]。

B-木の主な欠点は、キーを順番に辿るのが難しいことである。B-木では、内部ノードだけでなく、リーフノードにもデータポインタやキー値を置くことができるが、これは B-木の欠点であり、特定のレベルのノードを挿入する能力が低下するため、ノードレベルが増加し、良いことはない。

1.2　基本 B 木 (Basic-B-Tree)

この問題を解決するために B+木が提案された。B+木は、木の葉のノードにのみデータポイントを格納することで、先述した B-木の欠点を解消している。また、葉レベルのノードは互いにリンクされているため、データポインタの探索が容易で効率的であることも特筆すべき点である [5]。

B+木の変種について述べる前に、B+木の基本形となっている、B 木について挿入・探索・削除まで詳細を述べる。

B 木は、短いけれども重要な歴史を持っている。1960 年代の終わりに計算機メーカーと、独立した研究グループとが競い合って、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 H.Chiat や M.Schwartaz らが (Case Reserve 大学と共同で)、後述する B 木による手法を関連したやり方で検索や挿入を行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 (Stan ford 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.MacCreight 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 木と名付けた [3]。ここでは、基本 B 木のデータ構造と保守算法と

を、ノードから 2 本以上の路が出ることがない二分探索木を一般化したものとして紹介し、この後に個々の操作の費用について述べ、より深い B+木の理解に繋げていきたい。

二分探索木において、あるノードで選ばれる分岐は問い合わせキー (query key) とそのノード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。もし問い合わせキーの値か格納された値より小さければ左の枝が取られ、問い合わせキーの値より小さければ右の枝が取られる。図 1.1 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。

次に、図 1.2 に示されているような、個々のノードにふたつずつキーの入った、修正された探索きを考える。探索は、それぞれのノードでみっつの枝のうちひとつを選ぶことにより進行する。図 1.2 では、問い合わせ値が 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 42 と 81 の間の値の問い合わ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、葉に至る（失敗) まで、各ノードにおいて繰り返される。

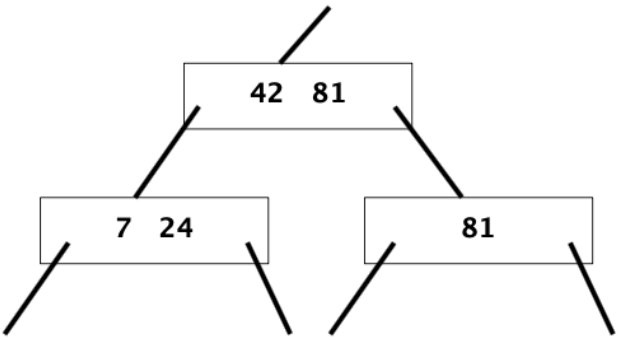


図 1.1: 従業員番号の二分探索木の一部, 太線は, 問合せ"15"に対してとられる路を示す

一般に、位数 d の B 木は図 1.3 に示すように最大 2d 個のキーと 2d+1 本のポインタを持

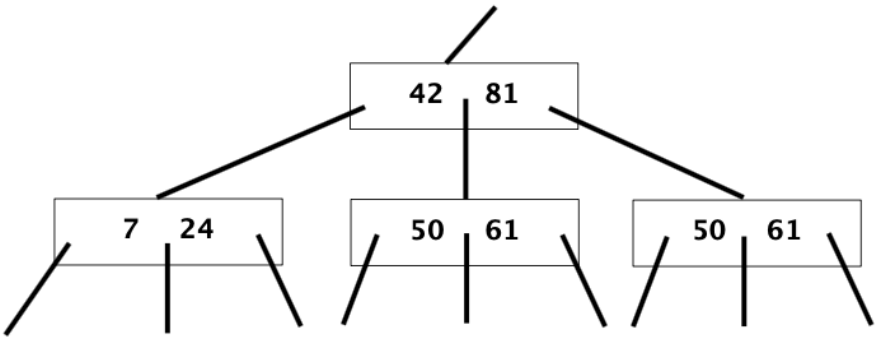


図 1.2: 個々のノードに 2 個のキーと 3 本のポインタをもった探索木

つ。実際にはキーの数とノードによっては異なりうるが、各ノードは少なくとも 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、個々のノードは少なくとも 1/2 は埋まっている。通常の実現方法では、ノード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、さらにノードの中にいくつか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ノード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。この後、ノードに 2 個以上のキーをもつと、検索・挿入・削除操作の費用がいかに安くなるかを調べる。

1.2.1 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う方法にある。二部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 1.4(a) に示すようなもので、長い路や短い路をもつが、一方図 1.4(b) に示すような均衡木では、同じ深さの所に全ての葉がある。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する。それ以外のノードは全て、少なくとも d 個以上のキーを持つ。) 直感的には、B 木は図 1.5 に示されるような形を持っている。n 個のキーを持つ B 木の最長路には、d を B 木の位数として、最大およそ log_n 個のノードがある。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うと、n 個のノード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_n 個より多くのノードを訪れることはない。ノードへの訪問は毎回二次記憶へのアクセスを陽数ので、木の均衡化については、多くの方式が提案されてきた。どの方式でも、均衡化を行うには多少の計算時間を要するため、検索操作の

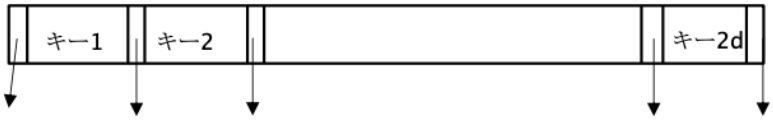


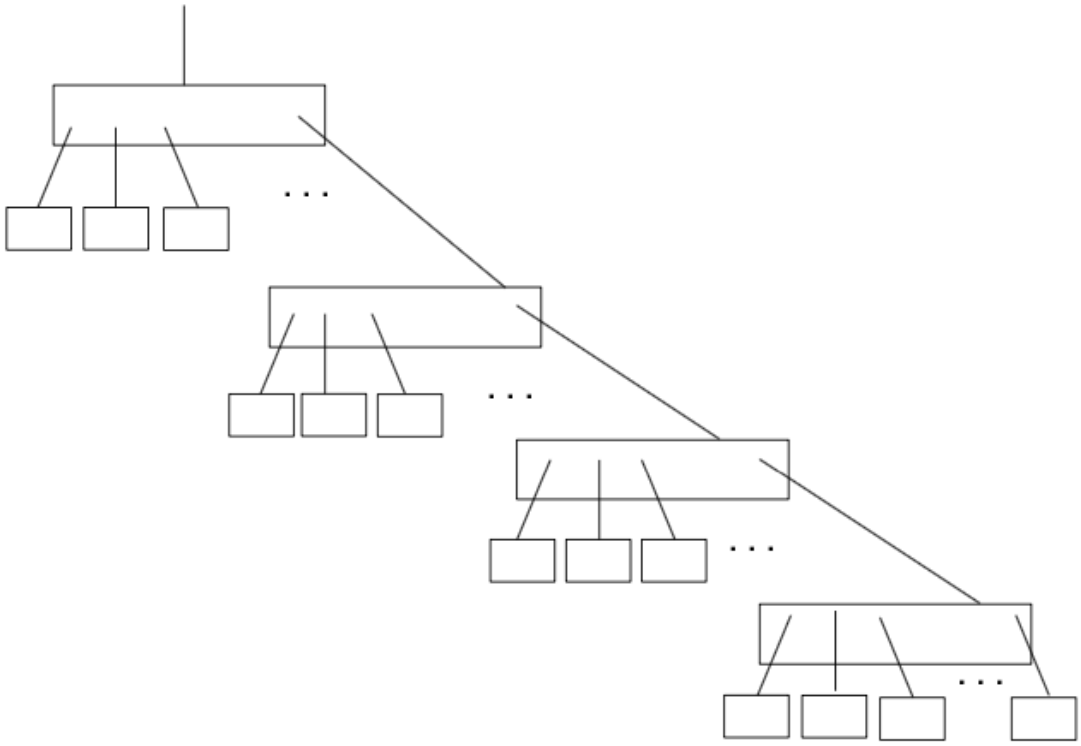
図 1.3: 2d 個のキーと 2d+1 本のポインタをもつ、位数 d の B 木のノード

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るために余分な記憶領域を用いている。そういうわけで、B 木は均衡化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。

1.2.2 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 1.6(a) の位数 2 の B 木を考える。位数 d の B 木のノードが d 個ないし 2d 個のキーを持つことから、この例におけるノードは 2 個から 4 個のキーを持つ。また図には描かれれていないが、現在のキーの数を憶えておくために、個々のノード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。続いて挿入が行われ、葉からへの方へと進行する手続きによって均衡が回復される。図 1.6(a) を例に取る。キー"57"を挿入しようとすると、左から四つ目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のキーを収容できるため、単に新たなキーが挿入され、図 1.6(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、面倒な事態が生じる。というのは、挿入に適した葉が既に一杯だからである。既に一杯になっているノード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、その場合ノードは図 1.7 に示すように分けられる。2d+1 個のキーのうち、小さい方から d 個がひとつのノードに置かれ、大きいほうから d 個がもうひとつのノードに置かれ、残った値は親のノードに昇進して分離地として役立てられるのである。通常の場合は、親のノードが追加キーを収容して、挿入処理が終了する。しかし、



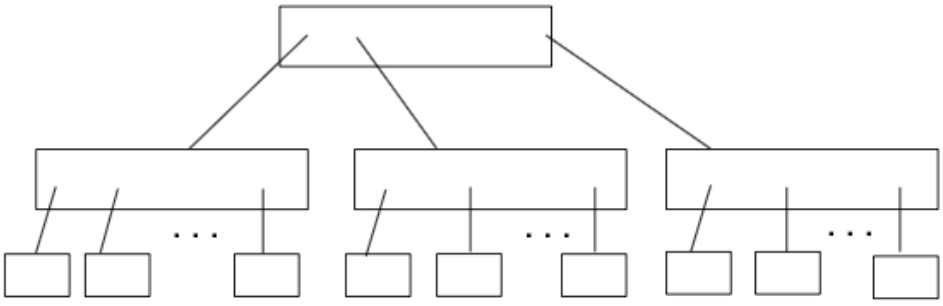


図 1.4: (a) 長い路を多数もつ不均衡木と,(b) 葉に至る全ての路の長さが全く等しい均衡木

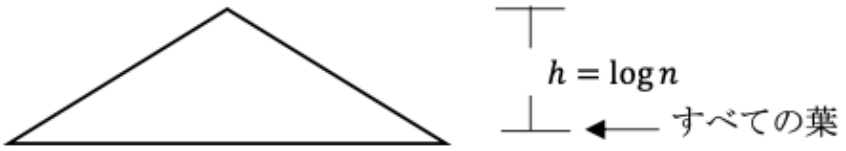


図 1.5: n レコードのファイルを索引づける、位数 d の B 木の輪郭

もし親のノード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われる。最悪の場合、分割は根まで伝播し、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。

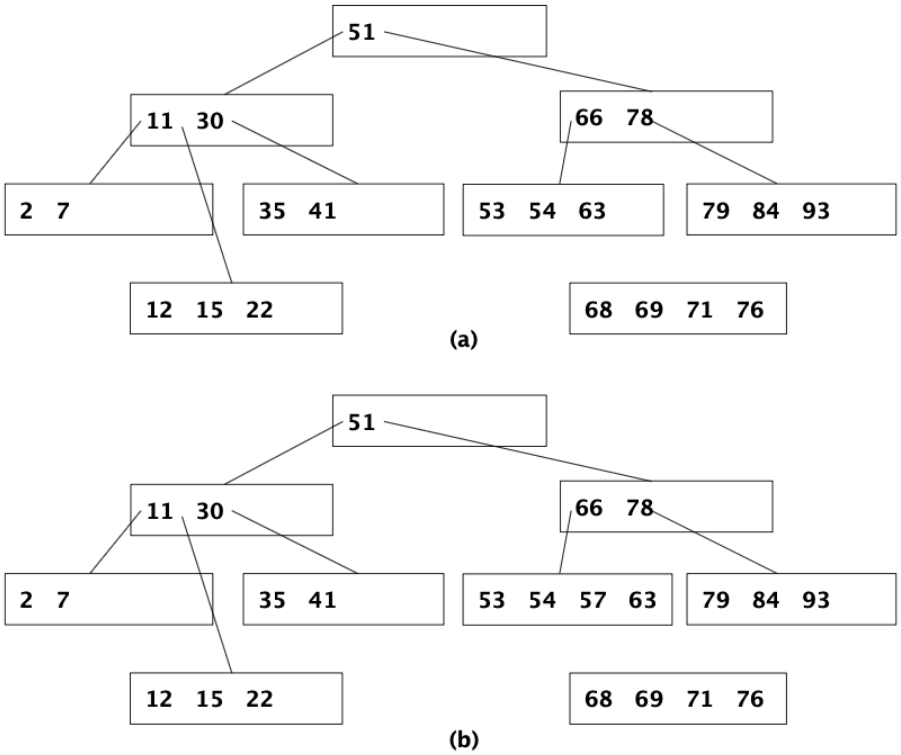


図 1.6: (a) 位数 2 の B 木と、(b) それにキー"57"を挿入したあとの木.

1.2.3 削除 (Deletion)

B 木における削除もまた、正しいノードを見つけ出すために検索操作を要する。検索が済んだ後は、二つの可能性がある。つまり削除すべきキーが葉にあったか、葉でないノードにあったかである。葉でないノードにおける削除では、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来なければならない。キー順で隣接しているキーを見つけ出すには、いま空いた区間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 1.8 がこれらの関係を示している。つまり、キー"17"を削除するときは、キー順で次に当たるキー"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ず、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 1.8 がこれらの関係を示している。

さて、いったん空き区間が葉へ「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（それと個々のノードは少なくとも d 個のキーをもつという B 木の性質) を取り戻すには、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ノードの間

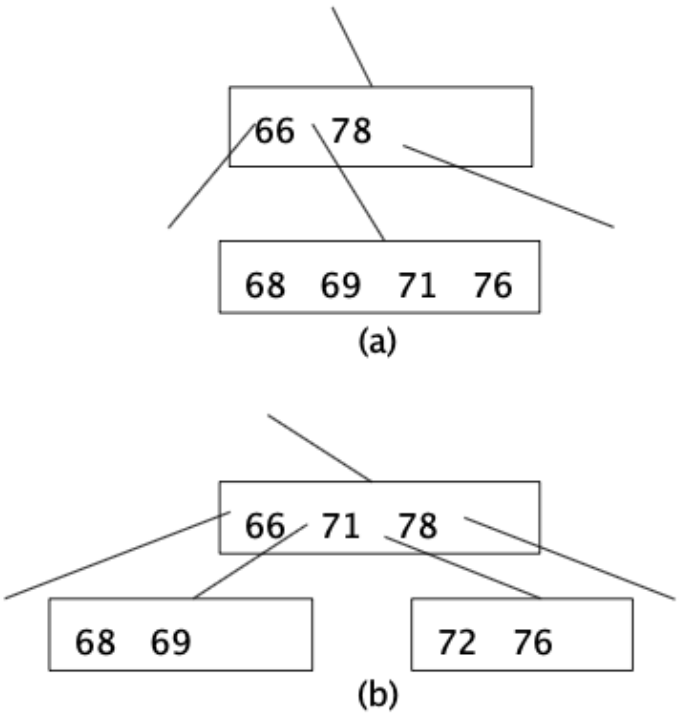


図 1.7: (a)B 木の葉とその祖先, (b) それにキー"72"を挿入した後の部分木.

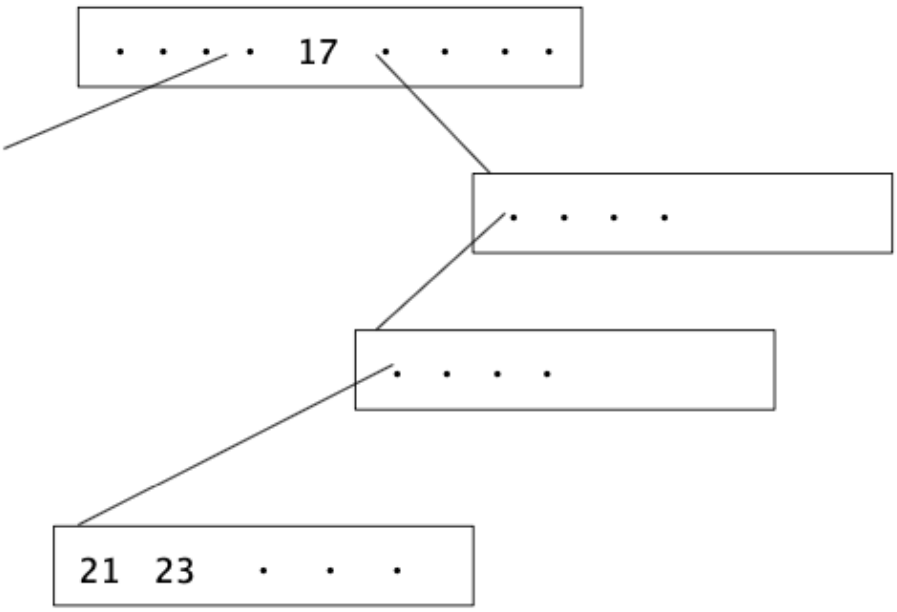


図 1.8: キー"17"を削除し、次にあたるキー"12"を探し出し、空いた場所に移動

で残されたキーを均等に分け合って、同一のノード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 1.9 に例示する。分離キー"50"の最終的な位置に注意。再配分によってノードの大きさを等しくすると、引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。

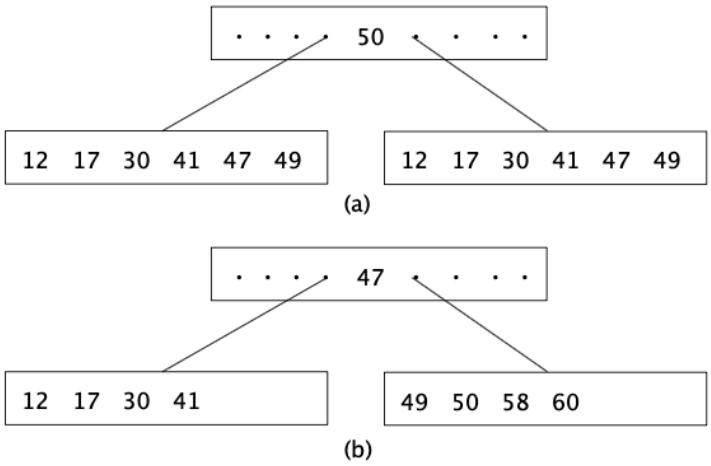


図 1.9: 隣りのノードとの間でキーを再配分する前 (a) と後 (b) の B 木の一部

もちろん、隣りどうしのキーの配分がうまくゆくのは、配分するきーが少なくとも 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ノードに統合され、もう一つのノードは捨てられる (連結は分割の逆であることに注意する)。ひとつのノードしか残らないので、先祖のノードにあって二つのノードを分離していたキーは不要になり、これもただ一つ残った葉に付け加えられる。図 1.10 は、連結と、分離の最終的配置の例を示すものである。

あるノードが、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、そのノードもまたアンダーフローを起こし、隣りのノードのうちの一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。

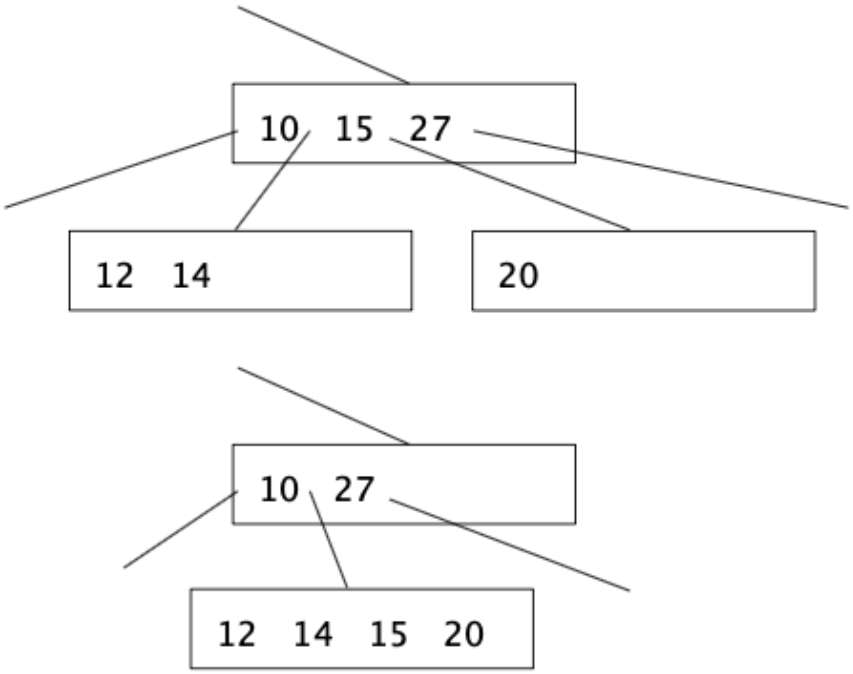


図 1.10: (a) 連結を引き起こす削除と、(b) 再均衡化された木

第2章　B+木

前述したようにB+木は、B-木を拡張したもので、検索、挿入、削除の操作をより効率的に行えるようにしたものである。B+木の葉のノードの構造は、B 木の内部のノードの構造とは全く異なっている。B+木は m 分木であり、ノードあたりの子ノードの数は可変であるが、多くの場合、子ノード数は大きい。B+木は根、内部ノード、葉からなる。根は葉であっても、2 つ以上の子を持つノードあっても良い。B+木は、各ノードがキー（キーと値のペアではない）だけを含み、リンクした葉を持つレベルが下部に追加されたB-木と見なすことができる

AVL 木は二分木をベースにしているが、これに対してB+treeは m 分木をもとにしたデータ構造である。m 分木とは、ノードが最大 m 個 ($m \geqq 2$) の子を持つことができる木構造で、二分木を一般化したものと考えることができる。二分木との対比から、m 分木のことを多分木（multi-way tree) とも呼ぶ。また、このような探索木を多分探索木（multi-way search tree) と呼ぶ。

m 分探索木のうち、次の条件を満たすものが m 階の B+木と呼ぶ。

1. 根は、葉であるか、あるいは 2〜m 個の子をもっている
2. 根、葉以外のノードは、「$\frac{m}{2}$」 m 個の子を持つ（「「x」は、x 以上の最小の整数、つまり、切り上げを表す）
3. 根からすべての葉までの経路の長さが等しい

条件 3 から、B+木はつねにバランスがとれていることになる。B+木の場合は、ノードが可変個の子をもてるという性質を利用して木のバランスをとる。たとえば、5 階の B+木を考えると、内部ノードは 3 5 個の子をもてることになる。この範囲内で子の数を調整して木を常にバランスがとれた状態に保つのである。

B+木の主な価値は、ブロック指向のストレージ（特にファイルシステム）において効率的に検索できるようにデータを保存することである。これは主に、二部探索木とは異なり、B+木は非常に高いファンアウト（ノード内の子ノードへのポインタの数、通常 100 以上のオーダー）を持ち、木内の要素を見つけるのに必要な I/O の回数を減らすことができるようにする為である。

図 2.1 は、高さ 2 で k=2 の B+木の例を示している。簡単にするために、自然数をキーとして使用し、関連する情報の表示を省略する。これは、B+木の構造に影響を与えないためである。全ての葉ノードが全く同じレベルにあり、ルートからリーフノードへの全てのパスの長さが同じであることが確認できる。これは、分割アルゴリズムの明らかな結果である [7]。

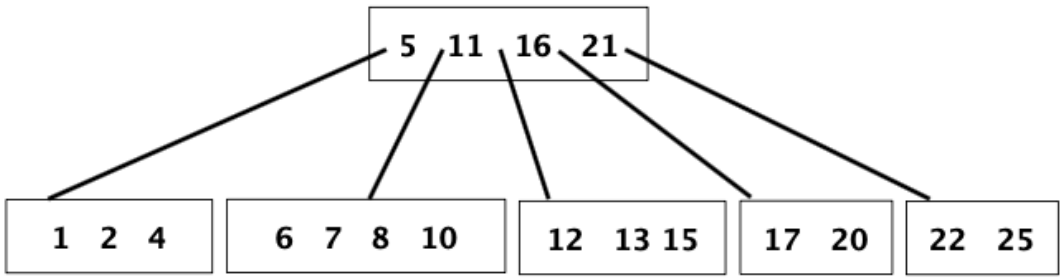


図 2.1: B+木

第3章　初出論文

B+木の概念について書かれた単一の論文は一つもないが、それぞれの論文でその概念が用いられている。具体的には、全てのデータをリーフノードに保持するという考え方は、興味深い変種として繰り返し持ち出されている。1972 年に発表された、Organization and Maintenance of Large Ordered indexs は、B-木とその派生データ構造のサーベイ論文で、特に B+木に十点が置かれている [10]。Douglas Comer は B-tree の初期の調査（B+tree も対象)で、B+tree が IBM の VSAM データアクセスソフトで使われていることを指摘し、1973 年に IBM が発表した記事を参照している [9]。

以下に、B+木について書かれた論文の内容を示す [11]。

B+木では、全てのキーが葉の中にある。B 木として編成されている上位レベルは、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 2.2 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、索引部のノードと葉のノードとは異なる形式でよく、さらに異なる大きさであっても構わない。特に、葉のノードは通常、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呼ばれる。シーケンスセットのつなぎは順処理を容易にする。

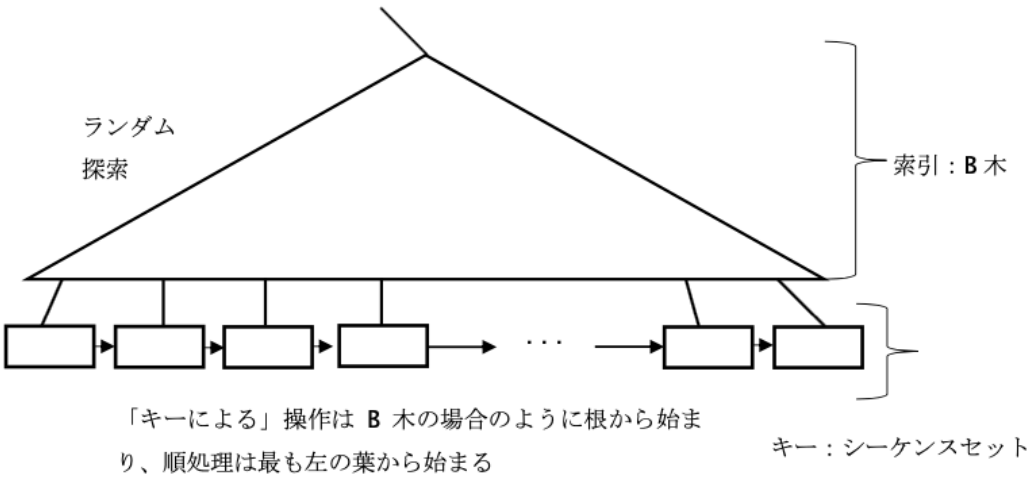


図 3.1: 索引部とキー部を別々にもった B+木

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しなければならない。まずは、検索操作を考えてみる。探索はB+木の根から始まり、索引を通って葉に至流。全てのキーが葉にあるため、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2.3は、消去されたキーの写しが、いかに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、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。

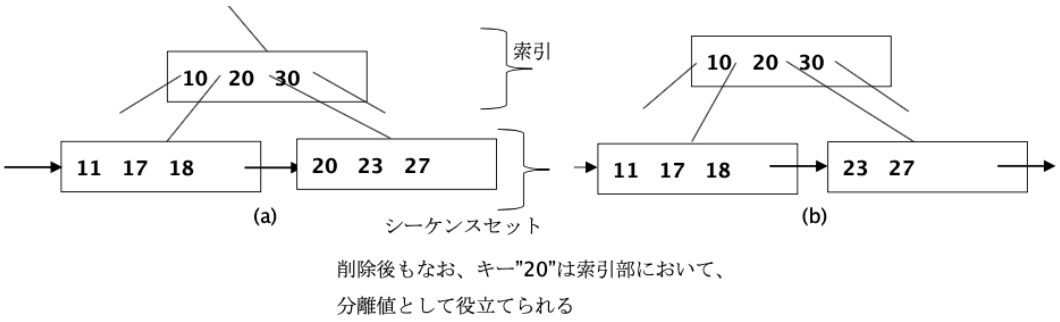


図 3.2: (a)B+木と (b) それからキー"20"を削除した後の B+木

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、はるばる葉に至るまで探索が進められるのである。

B 木は安い検索・挿入・削除を支援していながら、「次」をとる操作には log_n 回の 2 次記憶へのアクセスを要するかもしれない。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うのに高々1 回の 2 次記憶へのアクセスしか要しないという利点がある。さらに、ファイルの順処理の間はどのノードも 2 回以上アクセスされないため、主記憶ではわずかにノード 1 個分の場所が利用できさえすれば良い。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。

第4章　様々なB+木の変種

木は、もともと枝分かれしたものを表現するのに適したデータ構造である。ノードにデータを置くことにすれば、根から葉に向かって辿る経路が探索のプロセスを表していると考えることができる。なので、ノードに置いたデータと探索キーを比較しながら、根から葉に向かってたどっていけば、探索が実現できるはずである。木構造を利用した探索のアルゴリズムは数多く知られているが、どのアルゴリズムもこの考え方が基本になっている。

木構造を利用したアルゴリズムには、B+木以外にも、B+木の基本形となっている B 木、二分木、二分探索木、AVL 木、2-3 木、B*木、トライ（trie) などがあり、本節ではそれぞれについて述べていきたい。

4.1　二分木 (binary tree)

木構造の一種である二分木は、次のように定義される。

1. 空の木は二分木である
2. 次のいずれかを満たすノードからなる木は、二分木である
    - 子をもたない
    - 左の子のみをもつ
    - 右の子のみをもつ
    - 左右 2 つの子をもつ

この定義から、二分木はノードはたかだか 2 つの子しかもたないことがわかる。また、ノードが 1 つの子をもっている時、左の子であるか、右の子であるかを区別するのが、普通の木との大きな違いである。左の子を根とする部分木を左部分木、右の子を根とする部分木を右部分木という。

図 4.1 に二分木の例を示す。

(a) ではノード c はノード b の左の子であるが、(b) ではノード c はノード b の右の子となっている。したがって、(a) と (b) は二分木としては全く別のものである。二分木の図では、右斜め下向きの線で右の子を、左斜め下向きの線で左の子を結ぶようにして、両者を区別する。これに対して、普通の木では図 4.1(c) のようにノード b からノード c への線は真下に引く。

二分木を構成するノードは、最大でも 2 つの子しか持たない。そのため、右部分木と左部分木へのポインタをもつ構造体を使って、二分木を表現することができる。ノードを表す構造体は次のように定義する。

ここで、メンバ right が右部分木へのポインタ、メンバ left が左部分木へのポインタである。メンバ label はこのノードにつけたラベルである。また、メンバ right、left に NULL ポインタがセットされているときには、該当する子が存在しないことを意味する。図 4.2(a) の二分木は、図 4.2(b) のように実現される。図中の箱がノードを表している。箱は、左から順

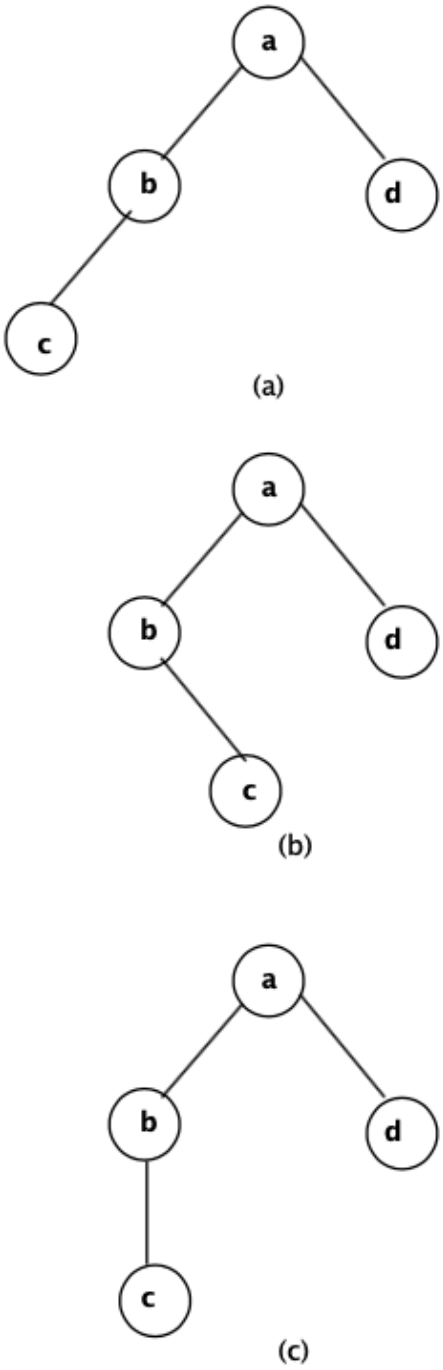


図 4.1: (a) 二分木 (b) 二分木 (c) 普通の木

```
struct node (
        struct node *left;
        struct node *right;
        mydata label;
);
```

にメンバ left,right,label にしきられている。ノード A の左の部分 (メンバ left) からノード B に向かって矢印がのびているが、これはノード B を指すポインタを表している。また、真ん中の部分（メンバ right) にはノード H を指すポインタが入っている。従って、ノード A の左の子はノード B、右の子はノード H であることがわかる。

4.2 二分探索木

木構造の一種として、二分木というもの上記で述べたが、これをもとにして探索を行うのが二分探索木（binary search tree) である。二分探索木とは、二分木の各節に要素（データ) を持たせたもので、

- 任意のノード x について、左部分木に含まれる要素はノード x よりも小さく、
  右部分木に含まれる要素はノード x よりも大きい

という関係が成り立っている。

二分探索木とはどのようなものか実際に見てみる。図 4.2 の（a) と (b) は、いずれも 7 つの要素 2, 5, 6, 7, 13, 15, 21 をもった二分探索木である。一般に、あるデータの集合を表現する二分探索木はいくつも存在する。また、このデータの集合を表す二分探索木は、図 4.2(a), (b) の 2 つの木以外にも数多く存在する。二分探索木の形は、要素を挿入する順番によって決定されるが、後ほど挿入の操作と一緒に考察する。

図 4.2 の (a) と (b) は、いずれも先ほど説明した二分探索木の条件を満たしている。例えば、(a) の木について考えてみる。根に注目すると、根そのものには 13 がセットされている。左部分木に含まれている要素は 2,5,6,7 で、いずれも 13 より小さいことがわかる。右部分木は 15,21 という要素をもっていて、いずれも 13 よりも大きくなっている。

また、要素 5 のノードに注目すると、左部分木には要素 2 が、右部分木には要素 6,7 が含まれていて、やはりこの関係が成り立っていることがわかる。この関係は、図 4.2 の全てのノードに成り立っている。

4.2.1 二分探索木の探索

二分探索木を利用した探索の手順は次のようになる。まず最初に、根の要素 x と探し出したいキーの値 k とを比較する。もし、$k = x$ ならば探し出したい要素は根に存在していて、探索は成功したことになる。$k < x$ ならば、値 k をもつ要素が存在するとしたら、二分探索木の性質からその要素は左部分木に含まれているはずである。また、$k > x$ であるならば、値 k をもつ要素は (存在するとすれば) 右部分木に含まれるはずである。つまり、$k < x$ なら左の子へ、$k > x$ なら右の子へ進む。また、進むべき子が存在しないーーつまり「行き止まり」に突き当たったーーのなら、キーの値をもつ要素は存在しない（探索が失敗した）ことになるので、この時点で処理を打ち切る。この手順を、キーの値をもつ要素が見つかるか、「行き止まり」に突き当たるまで繰り返す。

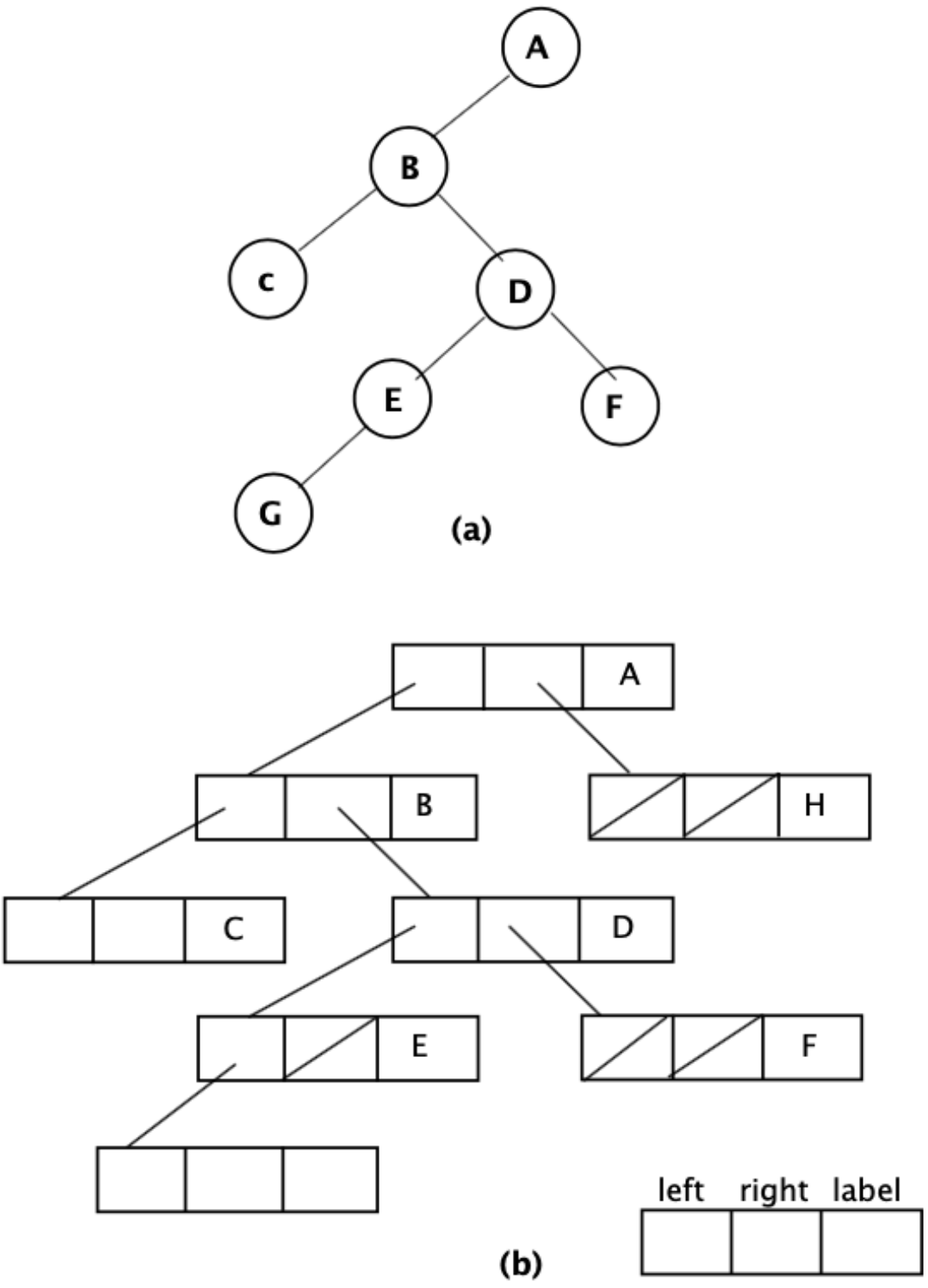


図 4.2: (a) 二分木 (b)(a) をポインタを使って表現したもの

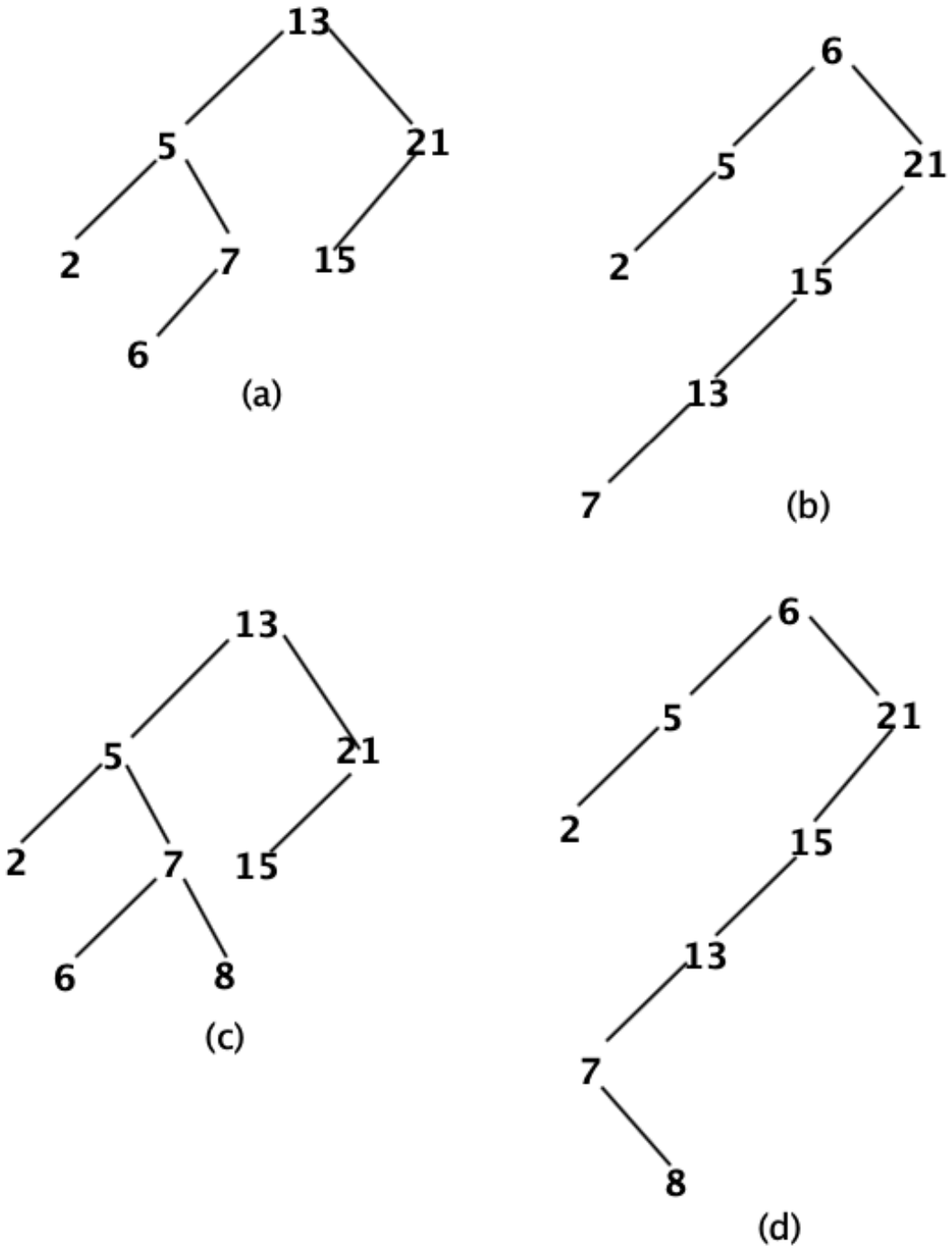


図 4.3: (a)(b) 二分探索木 (c)(a) に要素 8 を挿入したもの (d)(b) に要素 8 を挿入したもの

例えば、図 4.3(a) の二分探索木から、キーが 7 のデータを探索してみる。まず、最初に根に注目する。根の要素 13 はキーより大きいので、目指す要素は左部分木に存在する。従って、左部分木 (要素 5) に移動する。ここでは、要素の値が 5、キーの値が 7 で、キーの方が大きいので、右部分木（要素 7) に進む。今度は、キーの値を注目している要素の値が等しいので、探索は成功である。ここまでの過程をまとめると、次のように二分探索木を辿った事になる。

$13 \rightarrow 5 \rightarrow 7$

次に、図 4.3(b) の二分探索木を使って、今と同じデータ 7 を探索してみる。まず、根の要素 6 とキー 7 を比較すると、キーの方が大きいので、右部分木 (要素 21) に進む。次に要素 21 とキー 7 を比べると、キーの方が小さいので左部分木 (要素 15) に進む。ここでもやはりキーの方が小さいので、左部分木 (要素 7) に進む。ここまでやってきて要素 7 とキー 7 が一致するので、探索は成功となる。ここまでの過程をまとめると、次のように二分探索木を辿った事になる。

$6 \rightarrow 21 \rightarrow 15 \rightarrow 13 \rightarrow 7$

このように、二分探索木の性質さえ満たしていれば、その木がどんな形をしていても、探索が行える。このアルゴリズムでは、木のノードを辿りながらキーとデータの大小を判定して探索を行う。なので、その計算量は探索が成功（または失敗) するまでに立ち寄ったノードの数で表現されると考えられる。

4.2.2 二分探索木の挿入

次に、二分探索木へのデータの挿入について考えてみる。探索のときには、根から始めて二分探索木を下に向かって進んでいった。そして、キーと等しい値をもつノードに行き当たれば探索は成功、たどるべき部分木がなくなってしまえば探索は失敗としていた。ここで、見方を変えれば「辿るべき部分木 (子) がない」ということは、「本来ならそのデータはその部分木の位置に置かれるべき」ということを意味している。なので、二分探索木に要素を挿入するには、探索と同じ要領で木をたどり、辿るべき部分木がないという状態になったら、そこにデータを置けば良いことになる。

図 4.3(a) の二分探索木に 8 という要素を挿入してみる。まずキーを 8 として、探索をしてみる。根の要素は 13 で、キーの方が小さいので、左部分木 (要素 5) に進む。ここでは、キーの方が大きいので右部分木 (要素 7) に進む。ここでもキーの方が大きいので右部分木に進もうとするが、要素 7 には右部分木は存在しない。なので、要素 7 の右の子の位置に要素 8 を置けば良いことになる。

このようにして得られた木（図 4.3(c)) は、二分探索木の性質を満たしている。なぜなら、既に存在するノードの位置関係を保った上で、要素を二分探索木の性質を満たす場所に挿入しているからである。

同様にして図 4.3(b) の二分探索木に要素 8 を挿入すると、図 4.3(d) のようになる。図 4.3(c) と図 4.3(d) を比較してみると、同じデータの集合を表しているはずなのに、木の形がだいぶ違っていることに気づく。(c) の方が適度に枝分かれしていて、木の高さもあまり高くないのに対して、(d) の方はあまり枝分かれしないで、枝が伸びている。探索の計算量は、比較しなければならないノードの数で表現される。なので、(c) のように低くて十分に枝分かれした二分探索木の方が (d) よりも好ましいことになる。

4.2.3　二分探索木の削除

二分探索木からの要素の削除は、探索や挿入に比べると複雑である。何もせずにノードを削除してしまうと、二分探索木の定義を満たさなくなる可能性があるからである。

削除の手順としては、まず削除すべき要素を探し出す。この手順は、探索の場合と全く同じである。削除すべき要素が見つかったら、当然のことながら、それを取り除けばよい訳である。ここで、ノードが子を何個もっているかによって、場合分けして処理する必要がある。

最も簡単なのは、そのノードが葉である（子をもっていない) 場合である。この場合には、図 4.4 のように、ただ単にそのノードを取り去るだけで済む。

次に、子を 1 つだけもっている場合を考える。この場合には、削除されるノードがあった場所に、その子をもってくる。つまり、子を親の身代わりにすえるということである。なぜこれでうまくいくかを図 4.5 をもとに説明する。

図 4.5(a) の二分探索木から、要素 5 を削除しようとする。二分探索木の定義から、(要素 5 の親である) 要素 9 から見たとき、左部分木 (要素 5 を根とする部分木) に含まれる全ての要素は 9 より小さくなっている。この事実から、要素 5 の子は (左右どちらの子にしても)9 より小さいことになる。したがって、要素 5 がただ 1 つの子をもっているのなら、その子 (ここでは 3) を要素 5 の代わりにすえて得られる木は、やはり二分探索木の性質を満たす。

最後に残ったのは、削除すべき要素が 2 つの子をもっている場合で、このケースでは、削除するノードを、その右部分木に含まれる最小の要素で置き換えれば、二分探索木の性質を保つことができる (代わりに、左部分木に含まれる最大の要素で置き換えても良い)。例えば、図 4.6(a) の木から要素 7 を削除してみる。要素 7 の右部分木に含まれる最小の要素は 10 であるため、要素 7 にあったノードに 10 を移動する。また、要素 10 を取り去った場所には、要素 10 の子である要素 15 が繰り上がっていることに注意する (図 4.6(b))。

この手順が二分探索木の条件を満たすことは、先ほどと同様に簡単に証明できる。二分探索木においては、あるノード x の右部分木の任意の要素は、その左部分木のどの要素で置き換えても、左部分木に対して、二分探索木を満たしている。また、右部分木の最小の要素 m をノード x の場所に置けば、得られた木においては、新たに親となったノード m は右部分木のどの要素よりも小さくなり、右部分木に対しても二分探索木の条件は満たしている。

二分探索木の性質

二分探索木の特徴について考察する。二分探索木では、探索、挿入、削除の 3 つのどの操作についても、根から始めて木を辿って、然るべき位置を探し出す。つまり、挿入、削除を行うにも、まず探索をして挿入する位置、削除の対象を決めなければならない。そのため、二分探索木の処理の計算量は、探索の計算量に帰結されると考えて良い。

前述で考察したように、探索の計算量は木の枝ぶりによって大きく変化する。直感的に考えて、木が低くて枝分かれしているほうが探索は高速になる。図 4.7(a)、(b) の 2 つの二分探索木を比較してみる。どちらの木も、1 から 7 までの 7 つの要素をもっている。

図 4.7(a) のように、根から全ての葉までの経路の長さが等しいような二分木を完全二分木 (complete binary tree) と呼ぶ。実際には、完全二分木が作れるのは、ノードの数は $2^n - 1$ 個 (n は自然数) のときに限られている。ここでは制限をゆるめて、根から最も遠い葉までの経路の長さと、根から最も近い葉までの経路の長さの差が 1 以下である二分木を (広い意味での) 完全二分木と定義する。

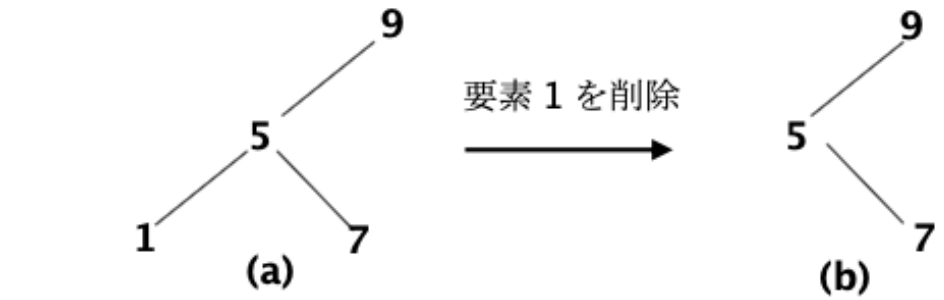


図 4.4: 子をもたないノード (葉) の削除

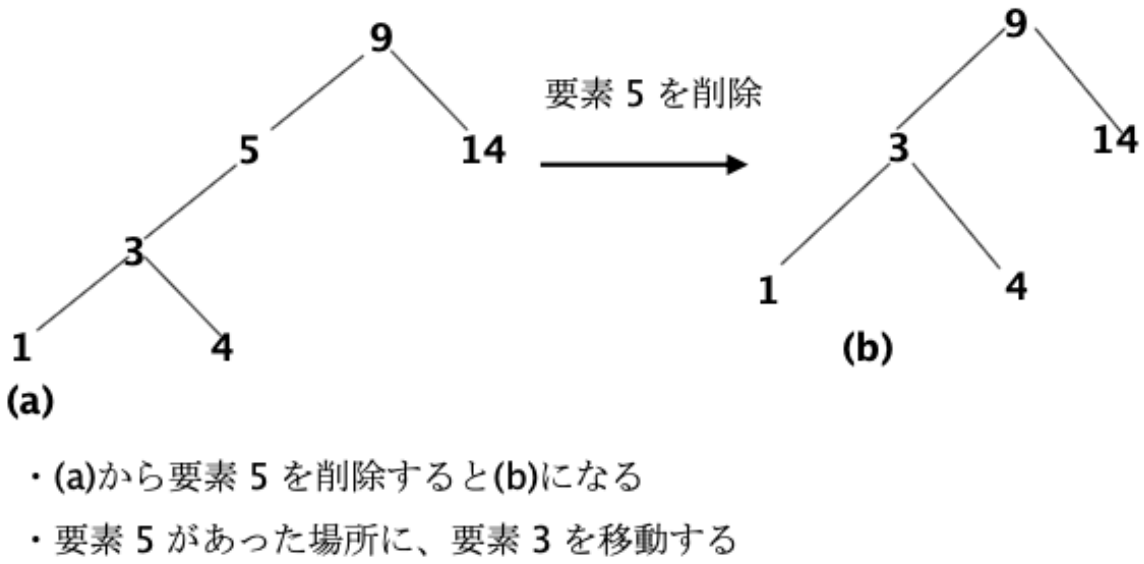


図 4.5: 子をもったノードの削除

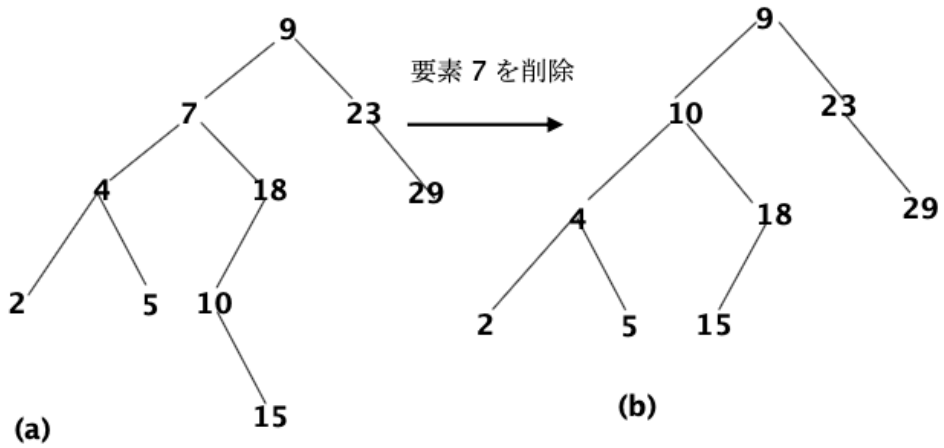


図 4.6: 子を 2 つもったノードの削除

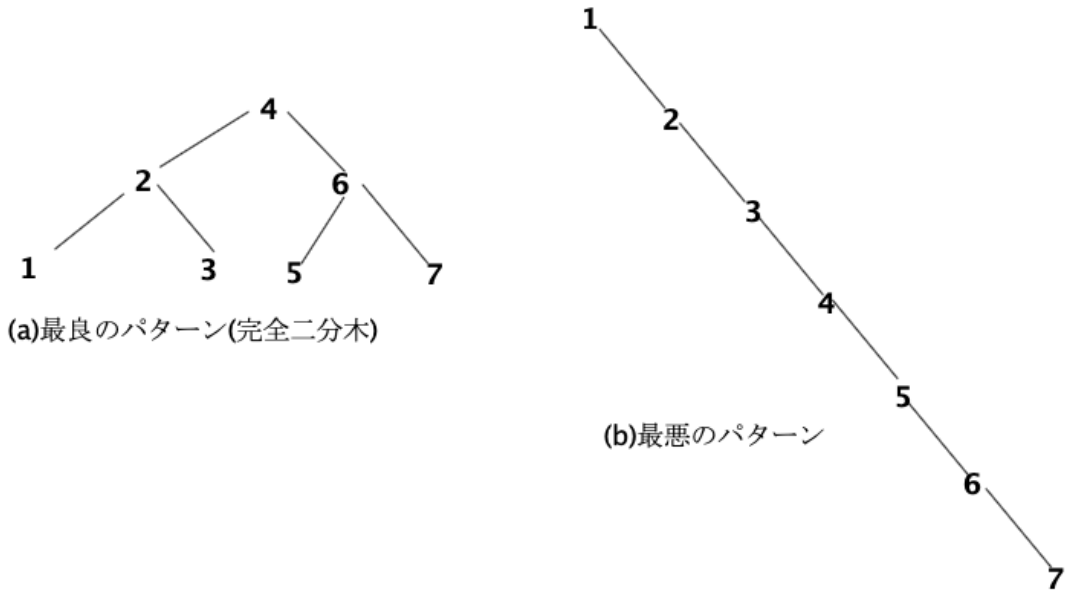


図 4.7: 二分探索木の最良と最悪のパターン

n 個の要素をもつ完全二分木は、根からノードへの経路の長さはたかだか $log_2 n + 1$ いないに収まる。また、このとき根からノードへの経路の平均長は $O(logn)$ となるため、探索も $O(logn)$ で行えることになる。二分探索木が完全二分木になっているとき、探索の効率は最もよくなる。

これに対して、図 4.7(b) も、同じ集合を表す二分探索木だが、要素が 1 列に並んでいるため、この木を探索することは線形探索を行うことと等価である。平均すると $n/2$ 個のノードを辿る必要があるため、探索の計算量は $O(n)$ になってしまう。

4.3 AVL 木

木の形が変わるときにーーつまり挿入・削除が行われるたびにーー木の形を見直して高さが $log_2 n$ 程度に収まるよう木を変形することが可能ならば、最悪の場合でも各操作が $O(logn)$ で実行できることを保証できる。このように、挿入・削除を行うたびに、形を変形して、高さが $log_2 n$ 程度に収まるようにした木を平衡木 (balanced tree) と呼ぶ。

4.3.1 AVL 木の考え方

最初に実現された平行木は、AVL 木 (AVL tree) と呼ばれるものである。AVL 木は、1962 年に Adel'son-Vel'skii と Landis が考案したもので、考案者の頭文字をとって AVL 木と呼ばれている。

AVL 木は、探索、挿入、削除の各操作が最悪でも $O(logn)$ で実行可能であるということを初めて示した点で歴史的な価値がある。しかし、現在では、B+木などのような性能の良い（計算量は同じだが、定数係数が小さいということ) 平衡木が知られているので、AVL 木の実用的な価値はあまりない。そのため、本節では AVL 木の考え方を簡単に説明していきたい。

AVL 木は、二分探索木に

- 全てのノードにおいて、左部分木と右部分木の高さの差が 1 以内に収まらなければならない

という制約を加えたものである。この制約によって、n 個のノードを持つ AVL 木の高さは $O(logn)$ のオーダーにとどまり、探索をつねに $O(logn)$ で実行することができる。

「各ノードにおいて、左部分木と右部分木の高さの差が 1 以内である」という条件を満たすには、具体的にどうすればよいのか、まずは挿入について考えてみる。二分探索木と同様の手順で、とりあえず要素を葉として木に挿入してしまう。次に「左部分木と右部分木の高さが 1 以内に収まっているか」を調べる。高さの差が 1 以内であれば、これ以上の操作は必要ない。もし、高さの差が 2 になっていれば、ノードを入れ替えて、左右の部分木の高さの差が 1 になるようにする (AVL 木の性質から、挿入を行う前には高さの差が 1 以内に収まっているため、挿入後の高さの差が 3 以上になることはあり得ない)。これが、AVL 木の基本的な考え方だが、実際にどのような操作を行うのかを、次で述べていく。

4.3.2 AVL 木の操作

図 4.8 のような AVL 木があったとする。図中の三角形は、部分木を省略したものである。ノード B の左部分木を X、右部分木を Y とする。また、ノード A の右部分木を Z とする。

ここで、部分木 X、Y、Z の高さは等しくなっている。これらの部分木の高さを h とする。ノード A から見ると、左部分木の高さは h+1、右部分木の高さは h となり、左部分木のほうが 1 だけ高くなっている。

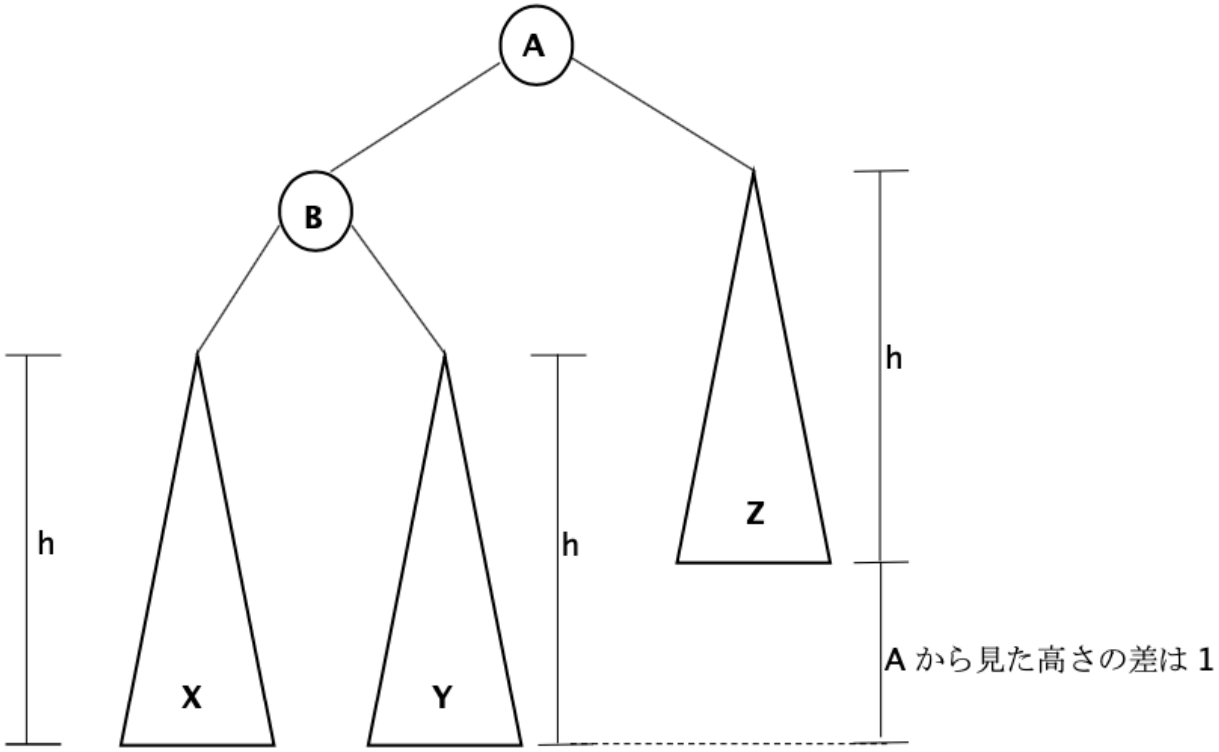


図 4.8: 挿入する前の AVL 木

ここで、ノード A の左部分木に要素を挿入した結果、左部分木の高さが 1 だけ高くなり、右部分木との差が 2 になったとする。このままでは左右の部分木の高さが 1 より大きいので、AVL 木の条件は満たされない。そのため、木の変形をして左右の部分木の高さを 1 以内に戻す必要がある。ここでは左部分木への挿入のみについて考察するが、左右を対称に入れ替えれば、右部分木への挿入についても同じことがいえる。

挿入した要素が、ノード B の左右どちらの部分木に挿入されるかによって、次の 2 つのパターンに分けられる。各パターンについて、別個に考えることにする。

- パターン 1：新しい要素は、ノード B の左部分木 X に挿入されて、部分木 X の高さが h+1 になった
- パターン 2：新しい要素は、ノード B の右部分木 Y に挿入されて、部分木 Y の高さが h+1 になった

パターン 1 の様子を示したのが、図 4.9(a) である。新しい要素を挿入した結果、部分木 X の高さは h+1 になり、ノード A から見ると左部分木が右部分木よりも 2 だけ高くなって

いる。ここで、図 4.9(b) のように、ノード A とノード B を入れ換えて、ノード B をノード A の親にする。この操作を、1 重回転 (single rotation) と呼ぶ。図 4.9(b) の木では、新たに親となったノード B から見ると、左右の部分木の高さはともに h+1 の高さの差は 0 となり、AVL 木の条件を満たしている。また、1 重回転によって、二分探索木の性質 (あるノードから見て、小さい要素は左部分木に、大きい要素は右部分木に含まれる) も保たれることに注意する。

次に、パターン 2 について考えてみる。パターン 2 では、図 4.10(a) のようになっている。新しい要素を部分木 Y に挿入した結果、ノード A から見て左部分木が右部分木よりも 2 だけ高くなっている。ここで部分木 Y をさらに分解して考えることにする。図 4.8 や図 4.10(a) では部分木 Y というブラックボックスとみなしていたものを、図 4.10(b) のように「ノード C と部分木 Y_1 と Y_2」という 3 つの部分に分けて考えるわけである。

部分木 Y の高さは h+1 なので、ちょっと考えると部分木 Y_1 と Y_2 の高さはともに h になるような気がするが、そうではない。新しい要素が挿入された方が h、そうでないほうが h-1 となる。しかし、ここでは「部分木 Y_1 と Y_2 の高さは、h か h-1 である」という点に興味がある。どちらが高さ h で、どちらが高さ h-1 であるかは議論の本筋には関係ない。

図 4.10(b) のノード A,B,C を入れ換えて、図 4.10(c) に示すように、ノード C を親にして、ノード A を右の子、ノード B を左の子にする。この操作を 2 重回転 (double rotation) と呼ぶ。新たに親になったノード C から見ると、左右の部分木の高さはともに h+1 と等しくなっている。2 重回転でも、二分探索木の性質は保存される。

回転による木のバランスの見直し/修正は、葉から親をたどりながら根に向かって上向きに伝播していく。例えば、図 4.11 のような木があって、ノード D の子として要素を挿入したとする。まずノード D について左右の部分木の高さの差が 1 以内に収まっているかをチェックして、もし必要であれば回転を行なって、バランスをとる。次に、ノード C について同様の処理を行う。さらにノード B、ノード A についても同様の処理を行う。ノード A についてバランスの見直し処理が完了すると、木全体が再び AVL 木の条件を満たすようになる。

4.4 B*木

階数の高い B+木は、外部記憶上での探索アルゴリズムとして優れている。例えば、1 ブロックが 1024 バイト長のディスク装置を使って、キー長が 10 バイトのデータを扱うことを考える。B+木のノードをファイルに格納すると、各ノードはファイルの先頭を基準にした位置 (オフセット) によって指定できる。このオフセットを 4 バイトで表現すると、1024/(10+4)=73 となり、1 ブロックに 73 階の B+木を格納できる。73 階の B+木が 3 段になっていると、33 万件程度 (73 の 3 乗) のデータを 4 回のディスクアクセスで探索できることになる。実際には、B+木にはかなりの空きがあるので、ここではうまくいかないが、かなりの高性能なデータ構造であることは確かである。

B+木の欠点として、内部ノードがもつ子の数が最小で [m/2] 個なので、最悪の場合には、記憶領域の約半分がムダになる、ということがあげられる。この点を改良したデータ構造として B*木がある。B*木は、ノードが持ちうる子の数を「2m/3」個以上、m 個以下としたものである。これを実現するためには、挿入のアルゴリズムを変更して、あるノード α がフルになったとき、まず隣のノード β に子をゆずるようにする。そして、ノード α、β ともにフルになった時点で、新しいノード γ を割り当て、ノード α とノード β がもっていた子 (合わせて 2m 個) を 3 等分すれば、最小で [2m/3] 個の子を持つ、という B*木の条件を満たすことができる。

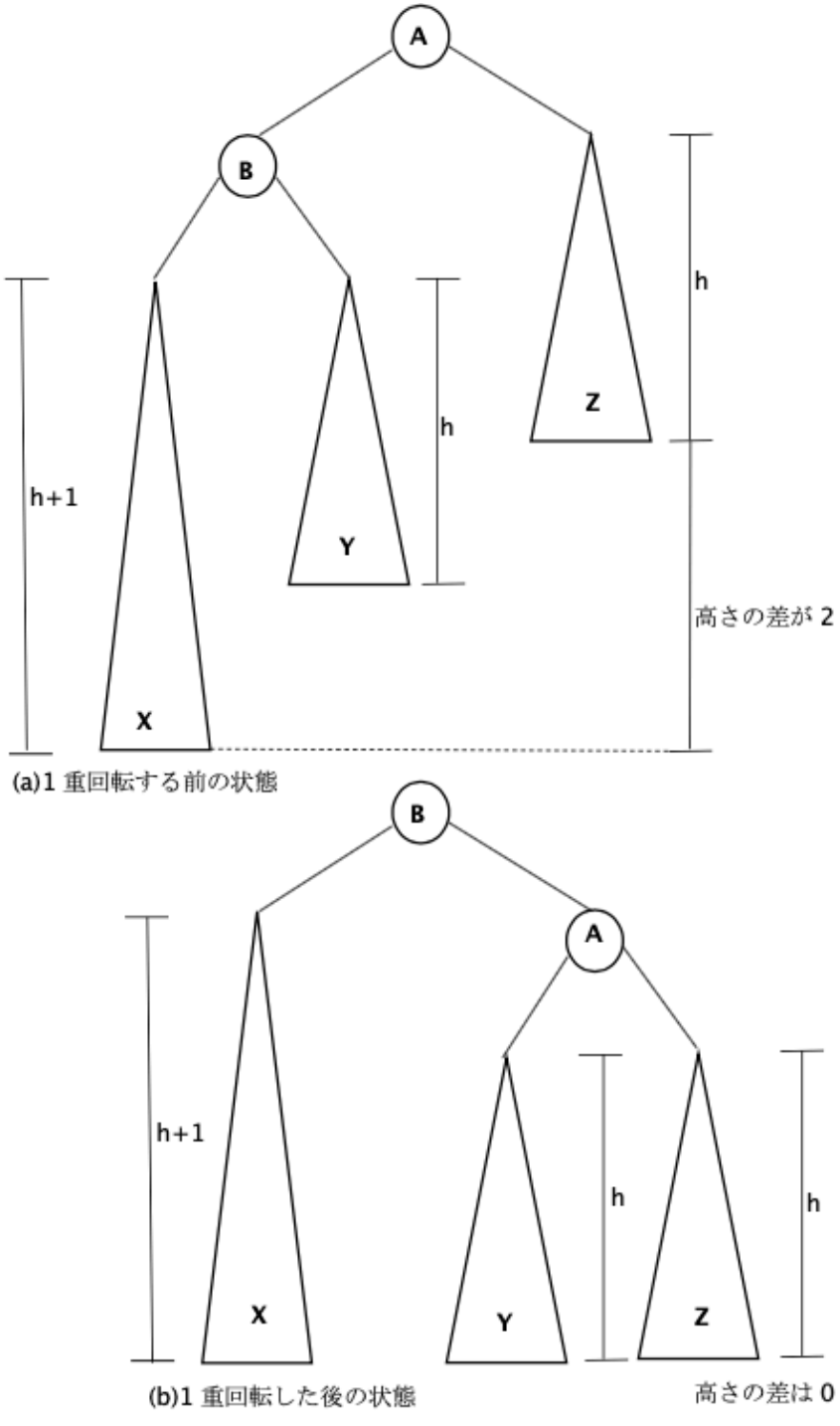


図 4.9: ノード B の左側への挿入 (1 重回転)

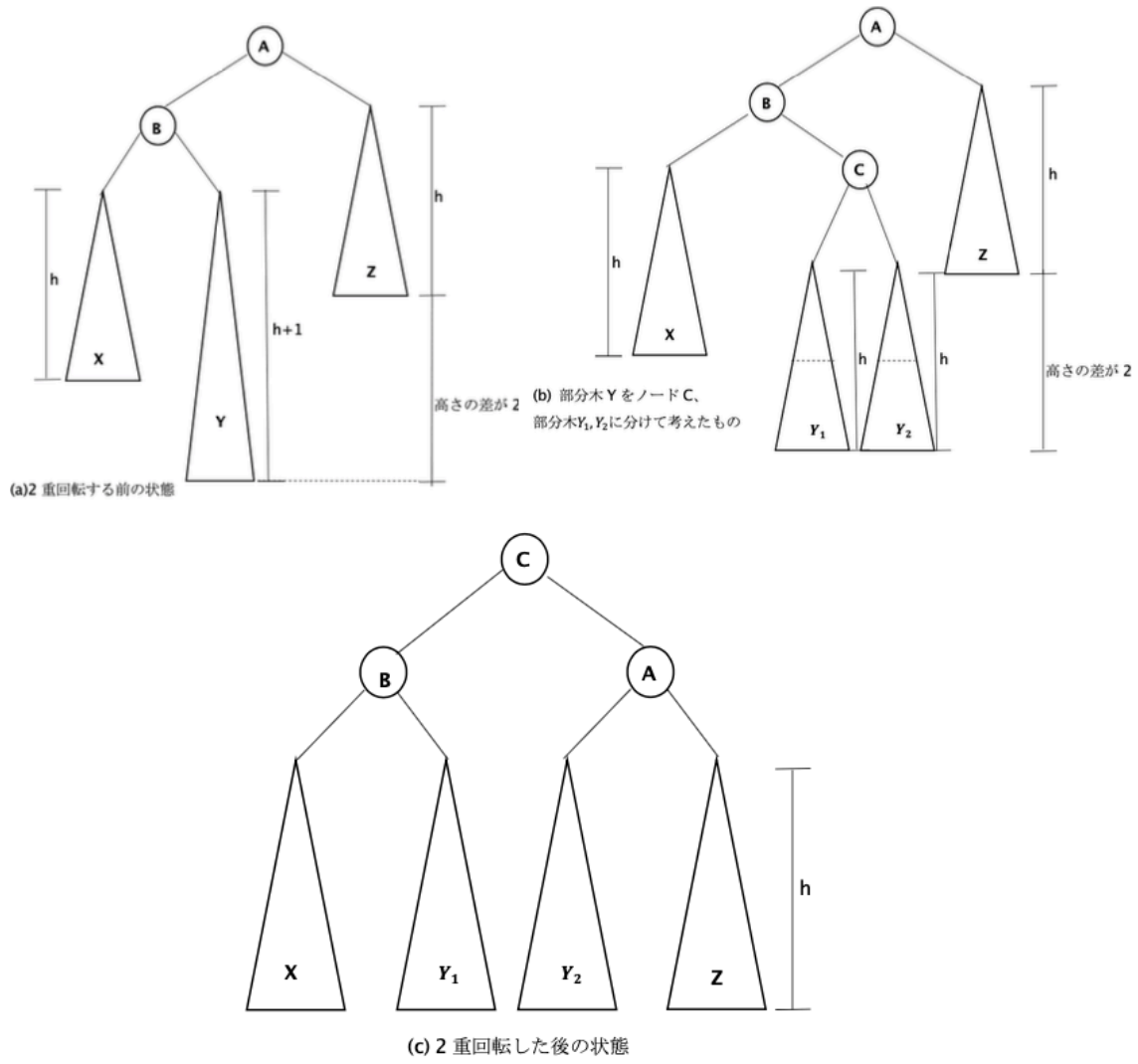


図 4.10: ノード B の右側への挿入 (2 重回転)

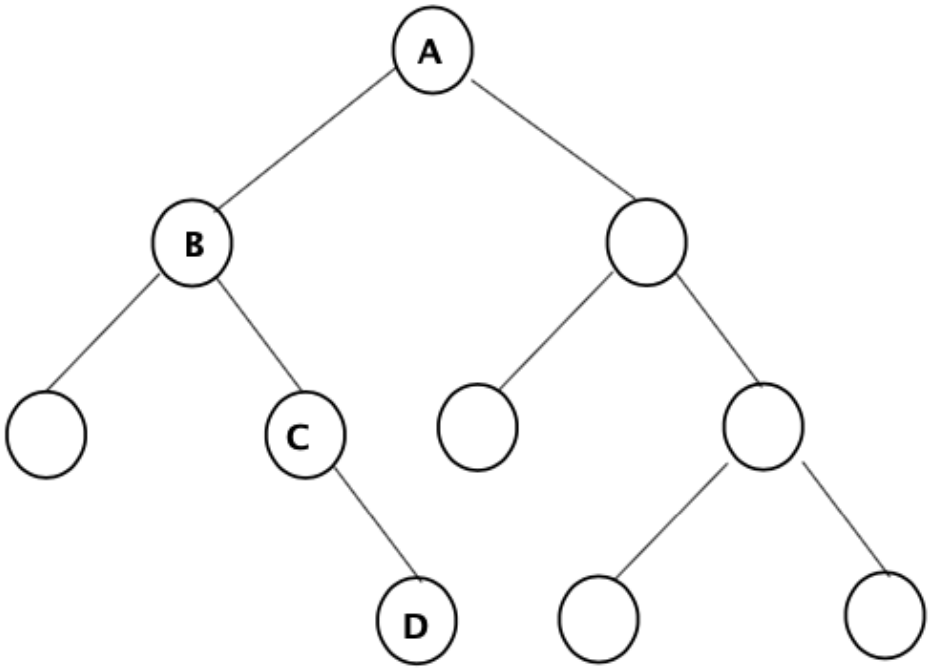


図 4.11: AVL 木における挿入の伝播

4.5 接頭辞 B+木 (Prefix B+tree)

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。そうすると、そこには実際のキーを入れておく必要はないのである。キーが文字列から成り立っているとき、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 [12] が、接頭辞 B+木という代案を考えている。

"binary"、"compiler"、"computer"、"electronic"、"program、"system"という一連の英字キーが図4.12のようにB+木に割り当てられていると仮定する。索引部における"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良い。例えば"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ため、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ノードに置け、分岐係数が増し、木の高さがへる。低い木の方が検索に費用がかからないため、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。先の例では、"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、分離の算法の都合の良い対を得るためにキーの近傍を調べまわることを提案している。こうするとノードにキーが不均衡に載ったままになるかもしれないが、ノードの一方に、二、三の余分なキーがあったとしても全体の費用に影響しないであろう。

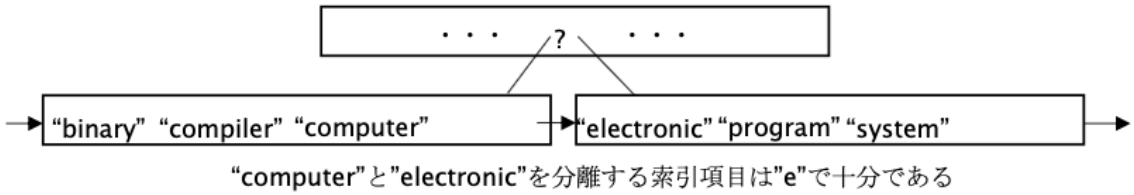


図 4.12: 圧縮ポインタをもったノード

4.6　2-3 木

2-3 木における個々のノードは 2 ないし 3 個の子をもつ（というのは 1 ないし 2 個のキーを持つからである)。従って 2-3 木の位数 1 の B 木であり、その逆も真である。2-3 木はノード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誠に適している。

第5章　設計と実装

1. 探索

B+木の探索は、探索キーの値と境目の値を比較することによって、どの部分木を辿れば良いかを判断することができる。まず、根から始めて探索キーを境目に値と比較しながら、部分木を下向きに辿っていく。そして、葉に到達したとき、探索キーの値と葉がもっているキーの値が等しければ探索は成功、等しくなければ探索は失敗。

B+木のおける探索の計算量は、根から始めて葉に到達するまでに $O(logn)$ 個のノードをたどる必要がある。また、各ノードでどの部分木を辿るか判断するが、この計算量は、境目の値と順番に比較すれば線形探索なので $O(m)$、境目の値が昇順に並んでいることを利用して二分探索を行えば $O(logm)$ となる。ここで、一般に n のほうが m よりもはるかに大きいので、m は定数とみなすことができる。なので、線形探索法、二分探索法のいずれにしても、判断の計算量は $O(1)$ となり、異なるのは定数項部分ということになる。従って、B+木における探索は、つねに $O(logn)$ で実行できることになる。

key がどこに収まるべきかを探すには、key の大きさを見ればわかる。左端から見ていき、key よりも大きい node->key[kid] を見つけたら、そこが収まるべき場所になる。探したいのはノードではなくリーフなので、node->isLeaf が True ではない場合は、再帰呼び出して更に下層のノードを見る作業に入る。こうすることで、key が収まるべき（key よりも大きい node->key[kid] がある) リーフ (node->isLeaf) が見つかる。なお、key よりも大きい node->key[kid] がなかった場合、全部探した結果右端のリーフに行き着く。findLeaf() の疑似コードを Algorithm 1 に示す。

Algorithm 1 key が収まるべき場所（リーフを探す）

```
1: function FINDLEAF((NODE *node, int key, int key)
2:     if (node->isLeaf)     return node;
3:     for (kid = 0; kid < leaf->nkey; kid++)
4:         if (key < node->key[kid]) break;
5:     return findLeaf(node->chi[kid], key)
6: end function
```

2. 挿入

探索によって、葉のひとつ前のノード a まで下りる。新しく作成した葉 β がノード a に追加する余地があるならば、葉 β をノード a の子として適切な位置に付け加えるだけである。

ノード a がすでに m 個の子を持っている場合は、ノード a を 2 つに分割することになる。

ノード a を 2 つに分割するということは、ノード a の親ノードから見れば、子が一つ増えたことと同様になる。そのため、ノード a の親ノードに対してもノード a と同じ処理を行う。

このようにして、ノードの分割や親へ親へと上向きに根に向かって伝播していく。途中でノードを分割しないで挿入できれば、そのノードの先祖については分割は伝播しない。

もし分割は根 γ まで伝播して、かつ根が m 個の子をもっていた場合は、根 γ をノード γ_1 とノード γ_2 に分割して、新たに根 σ を割り当てて、根 σ の子としてノード γ_1 とノード γ_2 を割り当てる。

B+tree の高さが増えるのは、要素の挿入によって、ノードの分割が根まで伝播した場合のみである [14]。

1. リーフノードに少なくとも 1 つのレコードを挿入しても、リーフノードがいっぱいになることがない場合、レコードを追加する
2. それ以外の場合、ノードをより多くの場所に分割して、分割した場所に key を適合させる。
   a. 新しいリーフを割り当て、ノードの要素の半分をツリー内の新たな位置に転送する。
   b. 二分木のリーフの最小キーとその新しいキーアドレスは、最上位のノードと関連付けられる。
   c. 最上位ノードがキーとアドレスでいっぱいになった場合、最上位ノードを分割する。
      - 同様に、ツリーの階層の最上位ノードの中央にキーを挿入する。
   d. 分割する必要のない最上位ノードが見つかるまで、上記の手順を実行し続ける。
3. 1 つの key と 2 つの指標からなる新しいトップレベルのルートノードを構築する [13]。

まず最初に、現在 NODE[0] に入っている key と今回新たに挿入された key を比較する。[0] の key よりも小さいものがあれば、新たな key は [0] に挿入しなければならないので、元々入っていた key を全て 1 個右にずらしている。そして、左端 [0] に key と data を入れる。

else の場合、現在の key をどこに差し込むべきか探さなければならない。その検索処理が 1 個目の for 文。見つけた時点で break しているので、i がその場所（挿入するべき場所）を示している。2 つ目の for 文で、i 以降の全ての key を右にずらす。insertIntoLeaf() の疑似コードを Algorithm 2 に示す。

具体的な例として、ここ t で、キー 13 と 19 が図 1 の B+木の挿入した場合を考える。これらは、B+木の構造を変更することなく、1 番目と 4 番目のリーフノードに配置される。次に、キー 9 をツリーに挿入すると、2 番目のリーフが分割され、中間のキー 8 がその親であるルートに移動され、ルートが分割される。結果の B+木を図 3 に示す。

Algorithm 2 leaf に新しくデータを挿入する

```
1:  function INSERTINTOLEAF((NODE *leaf, int key, DATA *data)
2:      if (key < leaf->key[0])
3:          for (i = 0; leaf->nkey; i > 0; i–)
4:              leaf->chi[i] = leaf->chi[i-1];
5:              leaf->key[i] = leaf->key[i-1];
6:          leaf->chi[0] = (NODE *)data;
7:          leaf->key[0] = key;
8:      else
9:            for (i = 0; i < leaf->nkey; i++)
10:             if (key < leaf->key[i]) break;
11:         for (int j = leaf-nkey; j > i; j–)
12:             leaf->chi[j] = leaf-chi[j-1];
13:             leaf->key[j] = leaf->key[j-1];
14:         leaf->chi[i] = (NODE *)data;
15:         leaf->key[i] = key;
16: end function
```

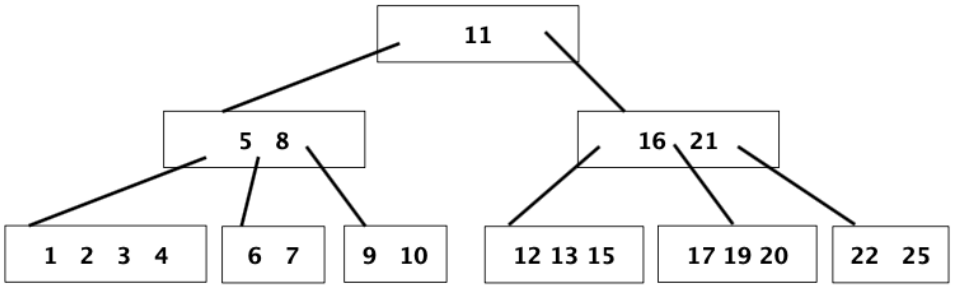


図 5.1: B+木

第6章　関連研究

6.1　Hashインデックス

B+木の比較として、Hashインデックスが挙げられる。Hashインデックスでは、Hash関数を使用し、キーのHash値とそのレコードの物理位置（TID:Tuple ID) を格納する。例えば、値を100で割ったときの余りを返す関数をHash関数とした場合、ID=234のHash値は34になり、この34に対応する位置にHash値とTIDを格納する。検索時も同様に、検索キーのHashインデックスを検索することで、目的のデータを取得する。

ツリー型のB-木インデックスでは、目的のキーを検索するために、木の根から順に木構造を辿るが、Hashインデックスでは Hash値を求めるだけで検索できる。したがって、Hash関数により上手くデータが分散している場合は、ほぼ定数時間 $O(1)$ で検索できることが、Hashインデックスの最大の特徴である。これは、検索だけでなく挿入、更新、削除でも同様である。しかし、Hashインデックスには、一致検索しか行えないという制約がある。不一致検索、範囲検索をすることはできない。また、検索結果の順序は保証されないため、ソート処理の手助けにもならない。 [15]。

6.2　空間インデックス

インデックスのアルゴリズムとして、空間 index というものもある。空間列にが多次元地理データが含まれているため、空間列をクエリするアプリケーションには、指定された空間関係を満たす全てのジオメトリを素早く識別する必要があり [16]、空間クエリからの検索結果を整理及び最適化するための空間データベースで使用される方法論として、空間インデックスを作成する。空間データベースは、通常のグリットベースのデータベースよりも当然複雑である。空間データベースは、オブジェクト間の関係を議論する時には3次元をうまく処理する必要があるという課題がある。 [17]。空間インデックス技術は、time-criticalなアプリケーションや空間ビッグデータの操作において中心的な役割を果たしている [18]。

6.3　並列処理とB+木

銀行、電子ショッピング、Webクエリ、図書館クエリなど多くのアプリケーションでは、データベース内での高度な並列処理が必要である。これには、並行処理と互換性が必要であり、ノンストップオペレーションが必要である。B+木は、次の理由から、このようなDBアプリケーションに最適である。

1. 全ての検索及び更新プロセスでルートにアクセスする必要があるが、ほとんどの更新でも読み取りが必要である。従って、ルート (及びその子の一部）は readが頻繁に行われる場所であるため、データシステムとオペレーティングシステムの標準的なキャッシュ技術によって常にメインメモリにキャッシュされる。これにより、非常に高速で高度な並列処理が可能になっている。

2. B+木のほとんどの更新と構造変換は、葉とツリーの下位レベルに限定されており、干渉はほとんどしない。

この処理シナリオを説明するために、様々な特殊な同期手法が読み取り及び更新トランザクション用に開発された。それらは、主にノードの粒度で様々なタイプのロックを処理し、B+木の特別な特性を利用するロックプロトコルに従う。そのような最初の同期プロトコルは、IBM の Reseach の System 用に開発された。

6.4 UB-trees for Multidimensional Applications

古典的なの B+木は、1 次元の線形順序の鍵空間用に設計されたものである。ここでの B+木は点クエリや区間クエリに優れている。しかし、多くのアプリケーションは、地理地図 (2 次元)、GPS データ (3 次元、緯度と経度に加えて高度があるため)、データウェアハウス (DWH）クエリーなど、多次元的なものである。

このような空間における範囲問い合わせは多次元矩形に対応し、その矩形内の多次元点を検索する必要がある。

このような多次元データ空間は、数学的変換により線形化され、通常の B+tree で表現することができる。このようなデータ構造をう UB-tree for Universal B-tree と呼んでいる。UB-tree は幾何学データベース、データウェアハウス、データマイニングのアプリケーションに有用である。

6.5 Masstree

Masstree は B+木とトライ木の性能を合わせた高性能なデータ構造である。2^64 のファンアウトを持つトライ木であり、トライ木のそれぞれのノードが B+木で構成されている。トライ木は prefix を共有させることによって長いキーを効率よく探索することができる。一方、B+木は短いキーにて対してはとも効率が良い。よってどちらの側面も持っている Masstre はより効率よく探索を行える [19]。

第7章　結論

B 木は安い検索・挿入・削除操作を支援していながら、「次」をとる操作には log n 回の二次記憶へアクセスを要するかもしれないという欠点がある。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うのに高々1 回の二時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ノードも 2 回以上アクセスされないから、主記憶ではわずかにノード 1 個分の場所が利用できさえすればいい。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。 [20]。

今後の課題

B+木は、B 木の一般的な変種として広く知られており、バルクデータに最も適したツリー型データ構造であると考えられている。しかし、大量のデータに対して B+木を構築するのは非常に時間がかかるため、並列 B+木システムの開発が必要であると考える。

並列 B+木によってデータ処理時間を短縮しつつ、効率的なインデックス構造を提供するための手法の検討が必要である。

謝辞

本論文の作成にあたり、終始適切な助言を賜り、また丁寧に指導して下さった川島准教授に深く感謝します。computer science やプログラミング能力がゼロの等しい状態かつ大学 4 年次から研究会に所属したにも関わらず、受け入れを許可して頂き、最後まで本当に暖かく見守って頂き、本当に感謝しています。川島先生のおかげで、computer science の面白さについて知ることができました。また、無知だった私でも勉強すれば社会に挑戦できるという自信と将来に対する可能性広げて頂きました。数学者である金沢准教授には、日頃から勉学の進み具合を気にかけて頂くだけでなく、人生に迷ったときに適切な助言と激励を下さり、感謝の念が絶えません。ここまで来れたのは金沢先生のおかげでおり、現在もこれからも私の恩師です。平素の研究会においても一年間に渡って大変お世話になりました。この場をお借りして重ねてお礼申し上げます。また、家族と友人には多大なる感謝を申し上げます。両親には 4 年間、経済的な支援をして頂き本当に感謝しています。私は、学業の面でも私生活の面でも迷いつまづき苦労することが多い 4 年間でしたが、常に家族と友人の支えがあり、どんな時も背中を押してくれたおかげで、勇気を持って進み続けることができました。最後に、論文執筆に協力して頂いた全ての方々に深く感謝致します。

関連図書

[1] "自律移動型ロボットの世界市場は 2027 年まで年平均成長率 19.6 %で成長すると予想される", https://www.jiji.com/jc/article?k=000004775.000067400&g=prt

[2] "B+-tree construction on massive data with Hadoop"https://www.researchgate.net/publication/319966450$_{B-tree_{c}onstruction_{o}n_{m}assive_{d}ata_{w}ith_{H}adoop}$

[3] "Machine organization of a growing search tree"https://www.mathnet.ru/eng/dan36593

[4] "B-tree"https://en.wikipedia.org/wiki/B-tree

[5] "B-tree"https://www.studytonight.com/advanced-data-structures/b-plus-trees-data-structure

[6] "B*tree"https://www.ueda.info.waseda.ac.jp/ ueda/articles/UbiquitousBTrees.pdf

[7] "B-tree"http://www.scholarpedia.org/article/B-tree$_{and_{U}B-tree}$

[8] "B+tree"https://en.wikipedia.org/wiki/B-tree

[9] "B+tree"https://dl.acm.org/doi/10.1145/356770.356776

[10] "Organization and Maintenance of Large Ordered indexes"http://www.inf.fu-berlin.de/lehre/SS10/DBS-Intro/Reader/BayerBTree-72.pdf

[11] "The Ubiquitous B-Tree"http://carlosproal.com/ir/papers/p121-comer.pdf

[12] "Prefix B-Trees"https://dl.acm.org/doi/pdf/10.1145/320521.320530

[13] "B+ TREE : Search, Insert and Delete Operations Example"https://www.guru99.com/introduction-b-plus-tree.html

[14] "B+木の概要と実装"https://tutuz.hateblo.jp/entry/2018/01/21/173057

[15] "第 2 回「Hash インデックス」 | NTT データ先端技術株式会社"https://www.intellilink.co.jp/column/oss/2018/051400.aspx

[16] "What Is Spatial Index?"https://www.ibm.com/docs/en/informix-servers/12.10?topic=data-spatial-index

[17] "What Is Spatial Index?"https://www.easytechjunkie.com/what-is-spatial-index.htm

[18] "DM-66 - Spatial Indexing"https://gistbok.ucgis.org/bok-topics/spatial-indexing

[19] "並行実行木 Masstree の調査"https://db-event.jpn.org/deim2017/papers/387.pdf

[20] "広くゆきわたったB木"https://www.ueda.info.waseda.ac.jp/ ueda/articles/UbiquitousBTrees.pdf