B-trees

2020年度前期・高度データ構造

本日の内容

B-tree

- 平衡探索木のひとつ.
- 要素の追加／削除を行っても高さを低く(要素数の対数程度に)保つことができる多分探索木.
- ブロック単位のアクセスが可能な外部記憶装置上に木構造を実装するのに適した構造.
- 実データベースシステムでも多用されている.

前回は、AVL tree について学びました。AVL-tree は高さが常に O(log n) で抑えられているような2分探索木でした。

本日は、B-tree について学びます。B-tree もまた高さが要素の対数程度で抑えられています。一方、AVL tree とは異なり、多分探索木です。

B-tree は実システムでもよく使われている有名なデータ構造です。

B-tree [Bayer & McCreight, 1972]

- B-tree は以下の特徴を持つ根付き木
  - 根は少なくとも 2 つの子を持つ.
  - 根と葉以外の各頂点は少なくとも d 個,
    高々 $2d$ 個の子を持つ (ただし $d \geqq 2$).
  - すべての葉は同じ深さである.
  - key(v): 頂点 v に格納されている整数の順序集合
    - $d - 1 \leqq |\text{key}(v)| \leqq 2d - 1$
  - $\text{child}_v[i]$: v の i 番目の子へのポインタ $(1 \leqq i \leqq 2d)$
  - 任意の $\text{child}_v[i]$ について,
    $\text{key}(v)[i-1] < \min(\text{key}(\text{child}_v[i])) \leqq \max(\text{key}(\text{child}_v[i])) < \text{key}(v)[i]$

B-tree は以下のような性質を満たす根付き木です。

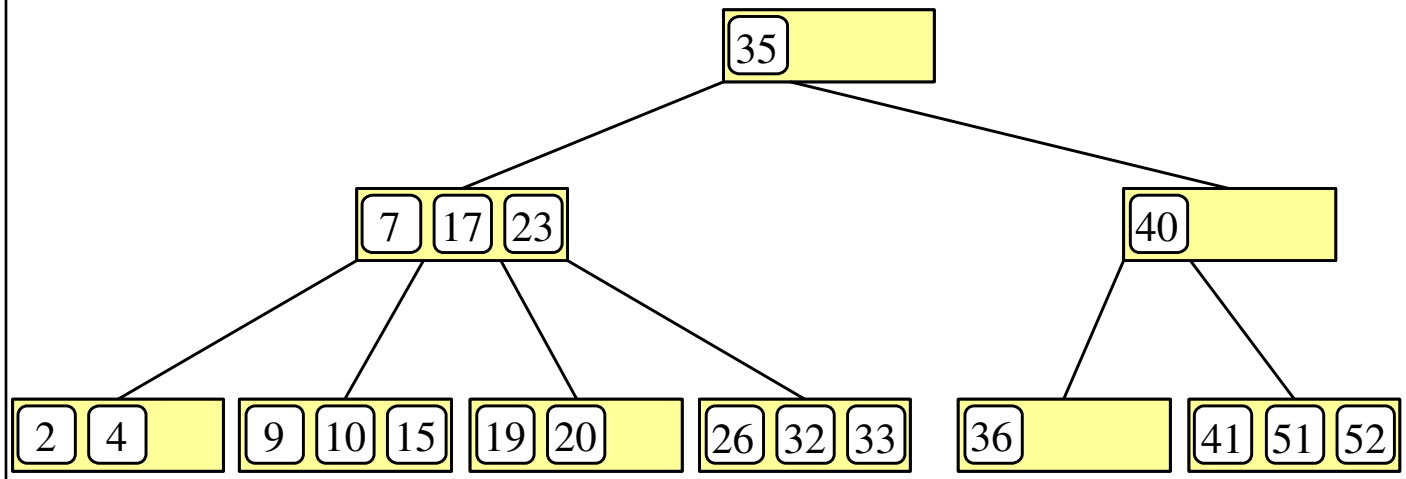
AVL tree とは異なり、d分木です（d ≧ 2）。ただし、各頂点の子の数は d ～ 2d の範囲です。

また、各頂点 v は key(v) という整数の順序集合を格納していて、その個数は d − 1 ≦ |key(v)| ≦ 2d − 1 となっています。

child_v[i] をこのように定義します。このとき、child_v[i] について、以下のような式が成り立ちます。これについては、次の具体例で説明します。

B-tree

- {2, 4, 7, 9, 10, 15, 17, 19, 20, 23, 26, 32, 33, 35, 36, 40, 41, 51, 52} に対する B-tree ($d = 2$)
  - 各頂点の子の数は2〜4個なので，2-4 B-tree と呼ばれる．



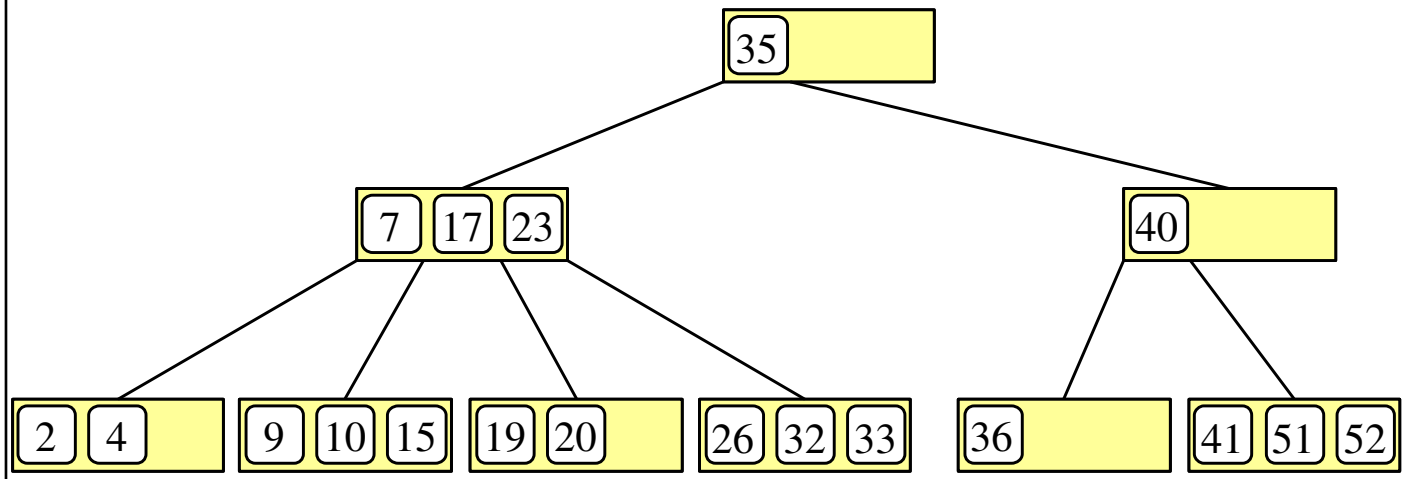
この整数集合に対する B−tree を図示しています。ここでは d = 2 としています。

黄色い長方形が各頂点で、その中の白い箱が各頂点 v が保持する key(v) の要素です。

各頂点から出ている辺が child_v[i] のポインタです。前ページの定義と合わせて確認してみましょう。

B-tree

- {2, 4, 7, 9, 10, 15, 17, 19, 20, 23, 26, 32, 33, 35, 36, 40, 41, 51, 52} に対する B-tree ($d = 2$)
  - 各頂点の子の数は2〜4個なので，2-4 B-tree と呼ばれる．



3ページの最後の式は、B-tree の以下のような性質を定式化しています。

7, 17, 23 を格納している頂点に注目してみましょう。

この頂点の左から1番目の辺を辿った先には 2, 4 があり、その辺を戻った先には 7 があります。

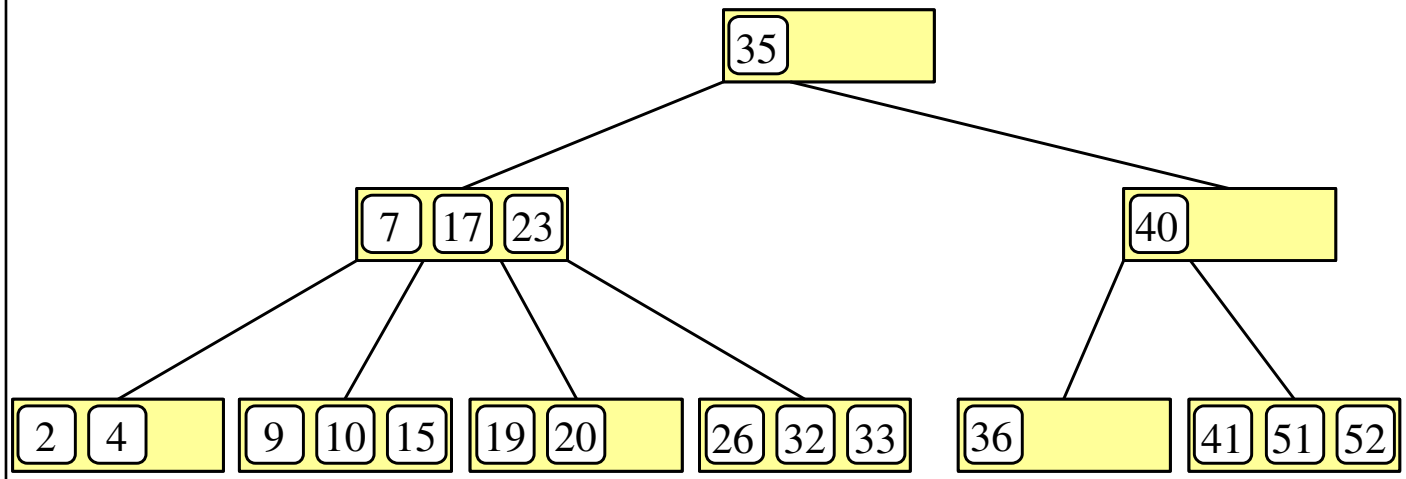
そこから次の辺を下に辿ると、9, 10, 15 があり、その辺を戻った先には 17 があります。

さらに、そこから次の辺を下に辿ると 19, 20 があり、その辺を戻った先には 23 があります。

最後に、4番目の辺を下に辿ると、26, 32, 33 があります。

B-tree

- {2, 4, 7, 9, 10, 15, 17, 19, 20, 23, 26, 32, 33, 35, 36, 40, 41, 51, 52} に対する B-tree ($d = 2$)
  - 各頂点の子の数は2〜4個なので，2-4 B-tree と呼ばれる.



さて、前ページで得られた数を、得られた順に列挙してみると、

2, 4, 7, 9, 10, 15, 17, 19, 20, 23, 26, 32, 33

となり、昇順に整列されていることがわかります。

この性質は、他の頂点についても成り立っています。よって、一般に、通りがけ順 (in-order) で B-tree を巡回すると、B-tree が格納している整数のソート列が得られます。

B-tree の高さ

定理 1

n 個の要素を持つ B-tree の高さ h について

$$h \leq \log_d \frac{n+1}{2}$$

が成り立つ.

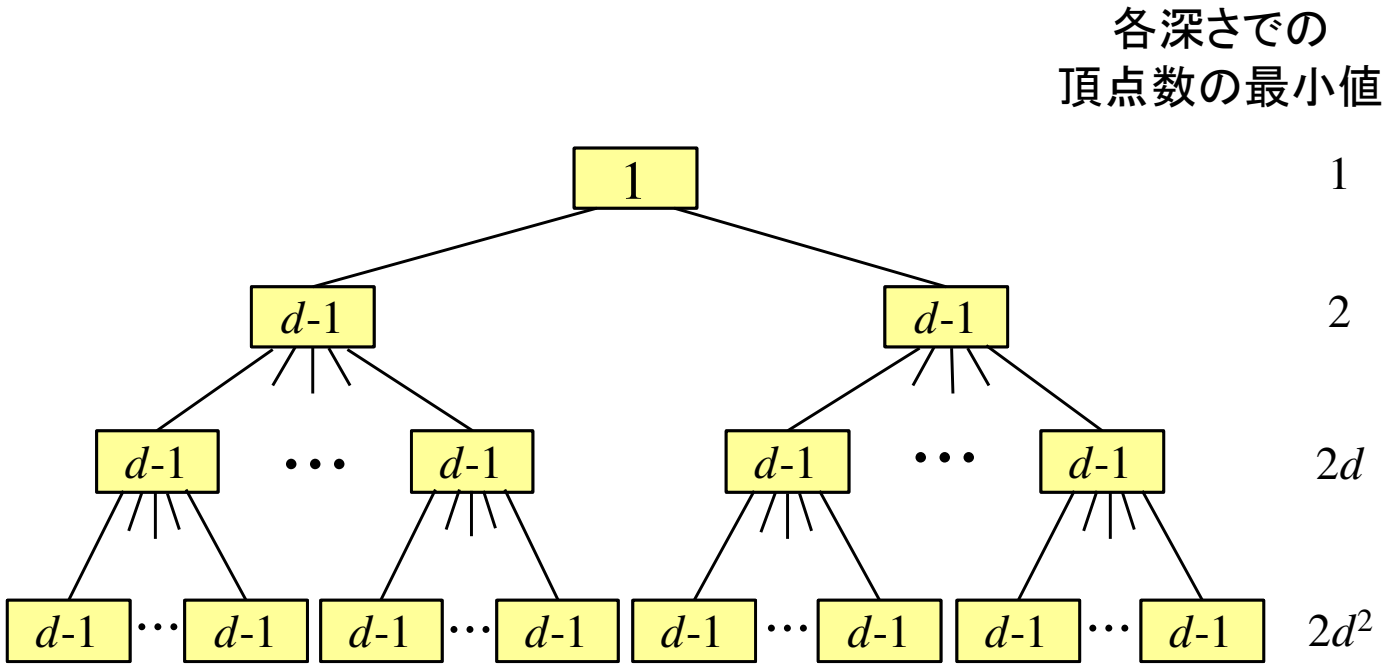
ここで、B-tree の高さについて、このような定理が成り立ちます。

n 個の要素を持つ B-tree の高さ h について、h は log_d ((n+1)/2) 以下となります。

この log の底は d で、d ≧ 2 であったことに留意しましょう。

次のページ以降で、この定理を証明します。

B-treeの高さ



この図は、各深さにおいて最も頂点数が少なる場合を示したものです。

上から見ていきます。まず、深さ 0 には根が1つだけ存在します。根は少なくとも2つの子を持つので、深さ 0 での頂点数の最小値は 2 です。

深さ 2 を見てみましょう。各内部頂点は少なくとも d−1 個の要素を格納しているので、子の数は少なくとも d です。よって、深さ 2 では、少なくとも 2 × d 個の頂点が存在します。

以下同様に、深くなるにつれて頂点の個数は少なくとも 2d^2, 2d^3, ..., 2d^{h−1} というようになっていきます。

B-treeの高さ

- 要素数 n に対して

$$n \geq 1 + (d-1)\sum_{i=1}^{h} 2d^{i-1}$$

$$= 1 + 2(d-1)\left(\frac{d^h - 1}{d-1}\right)$$

$$= 2d^h - 1$$

➔ $h \leq \log_d \frac{n+1}{2}$

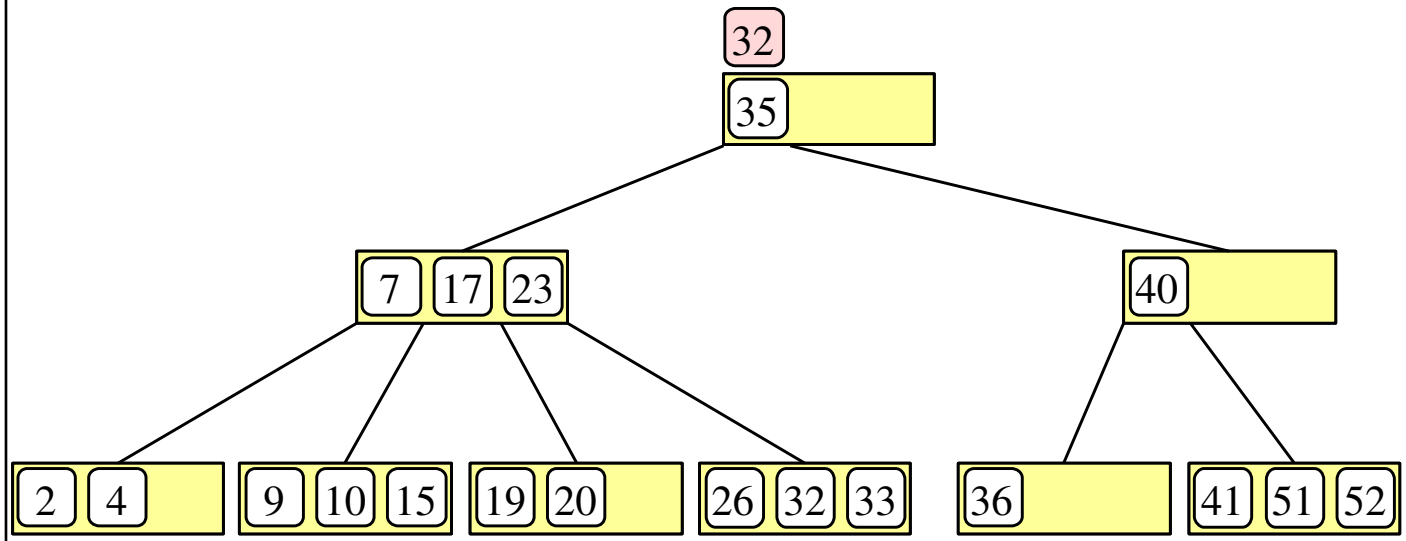
前ページの図から、要素数 n の集合に対する B-tree について、このような不等式が成り立ちます。ここで、(d-1) は、深さ 2～h の各頂点が格納している要素の個数です。

右辺を整理すると、2d^h － 1 が得られます。両辺の対数を取ると、h ≦ log_d ((n+1)/2) が得られました。

以上で証明終わりです。

member on B-tree

- member(32, *S*)



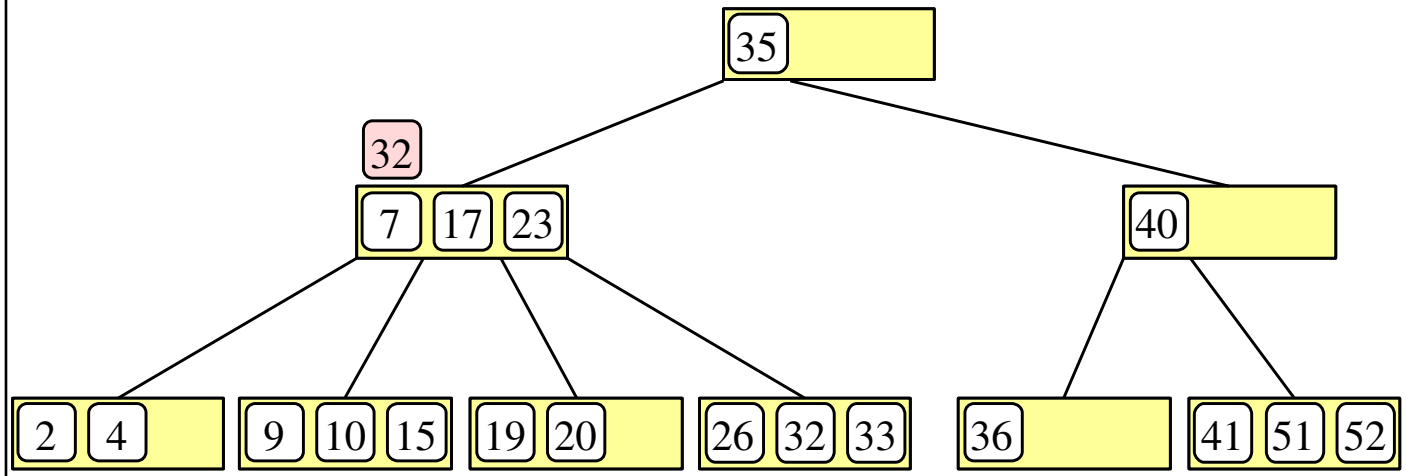
次に、B-tree を用いて member クエリを処理する方法を説明します。

この B-tree を用いて、32 を探すことにします。まず、根から始めます。

ここでは、2分探索木と同様に、32 と 35 を比較します。32 < 35 なので、根の左の子に移動します。

member on B-tree

- member(32, *S*)

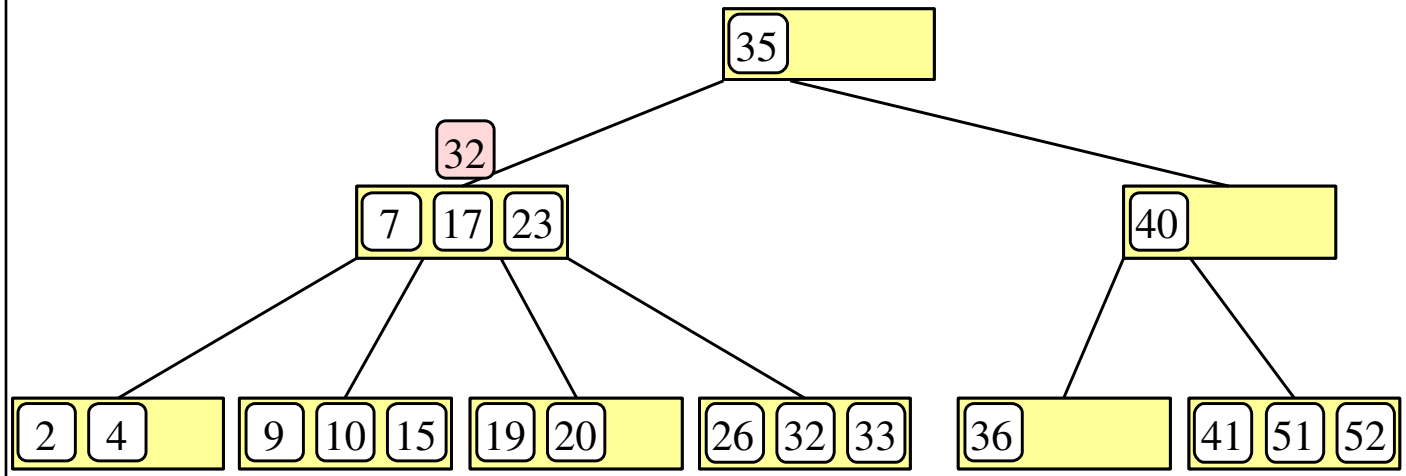


ここで、この頂点に格納されている3つの値と、32 をそれぞれ順番に比較していきます。

32 > 7 なので、次の要素と比較します。

member on B-tree

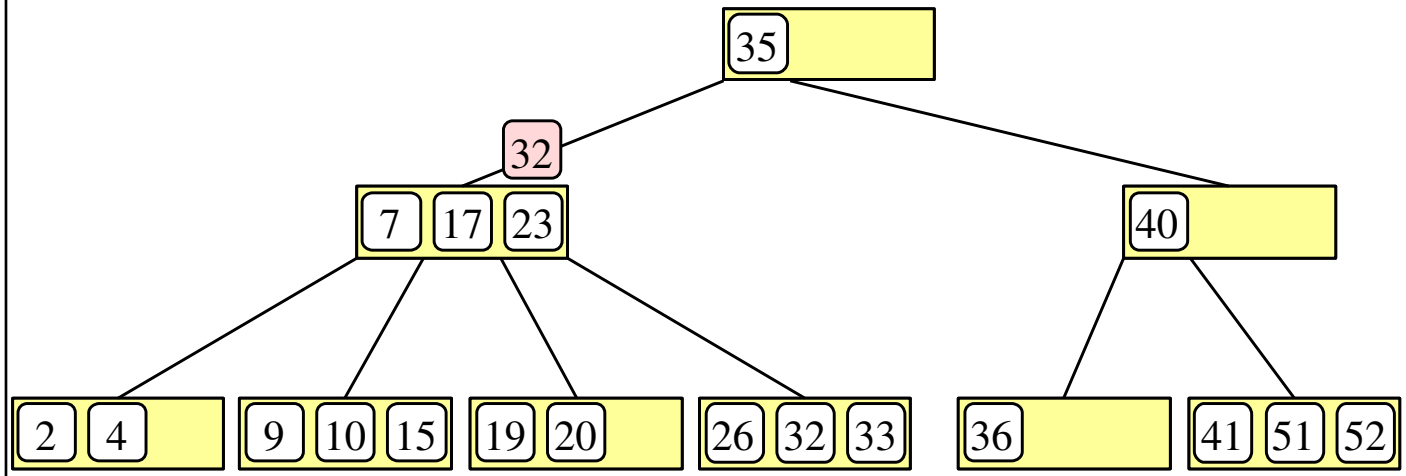
- member(32, *S*)



32 > 17 なので、次の要素と比較します。

member on B-tree

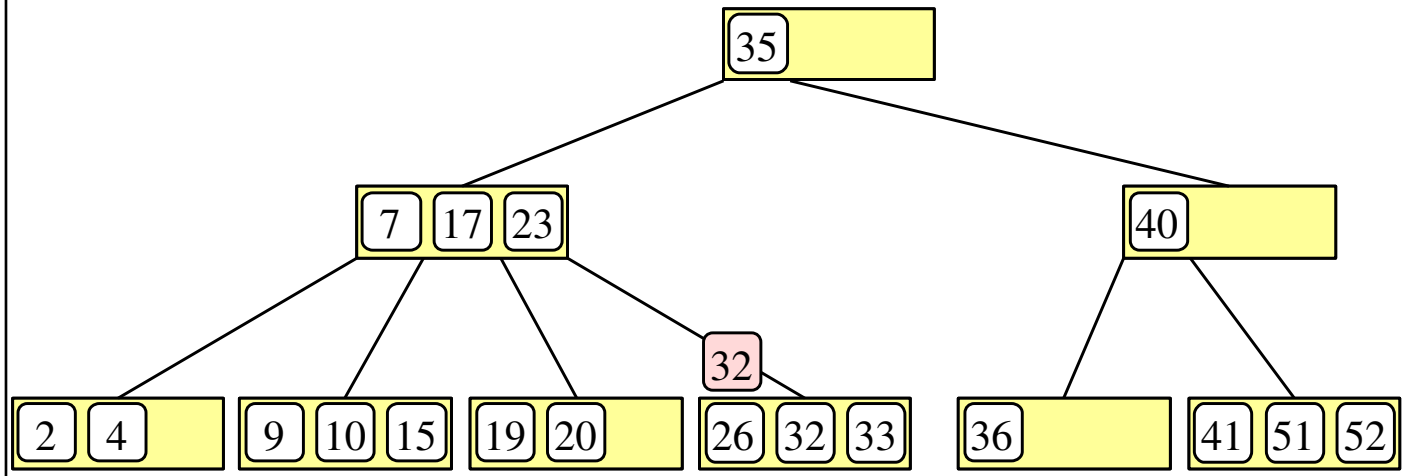
- member(32, *S*)



32 > 23 で、かつ 23 がこの頂点の最後の要素なので、一番右の辺を辿って子に降りていきます。

member on B-tree

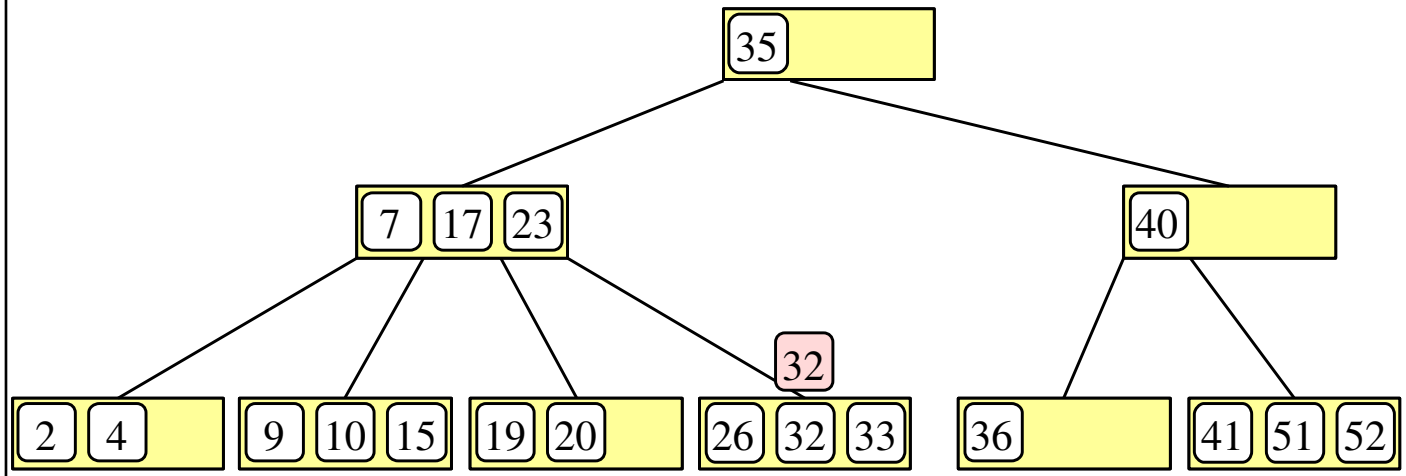
- member(32, *S*)



32 > 26 なので、次の要素と比較します。

member on B-tree

- member(32, *S*)



ここで 32 = 32 となり、32 を見つけることができました。

B-tree では、このような方法で member クエリを処理します。

member on B-tree

定理 2

B-tree を用いて member を $O(d \log_d n)$ 時間で計算できる.

【証明】

- 定理 1 より，B-tree の高さは $O(\log_d n)$
- 各頂点において，要素を線形探索➔ $O(d)$ 時間

B-tree を用いた member クエリの時間計算量について、この定理が成り立ちます。

先ほど解説した方法を用いた member クエリの時間計算量は O(d log_d n) です。

証明します。前述の定理1より、B-tree の高さは O(log_d n) でした。

探索中に訪れる各頂点において、いま探している数と、その頂点に格納されている要素を線形比較していました。各頂点は d-1～2d-1 個の要素を格納しているので、各頂点における時間計算量は O(d) です。

これらを掛けて O(d log_d n) が得られます。

証明は以上です。

演習問題 1

- B-tree の各頂点における探索を工夫することで，member の時間計算量を $O(d \log_d n)$ から $O(\log n)$ に改善できる．その方法を示せ．

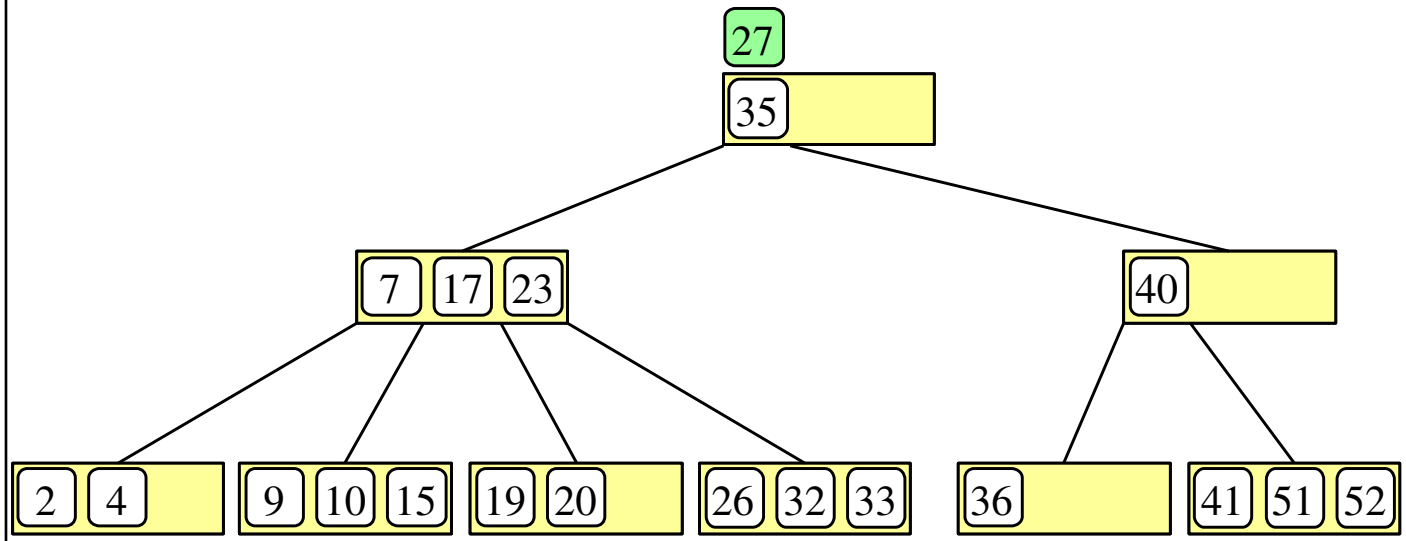
※ 演習問題 **提出〆切：６月４日（木）２３：５９**
pdf の作成方法について、最終ページを参照のこと

では、本日の演習問題（その１）です。

B-tree の member クエリの時間計算量を O(log n) に改善する方法を示してください。

insertion on B-tree

- insert(27, *S*)



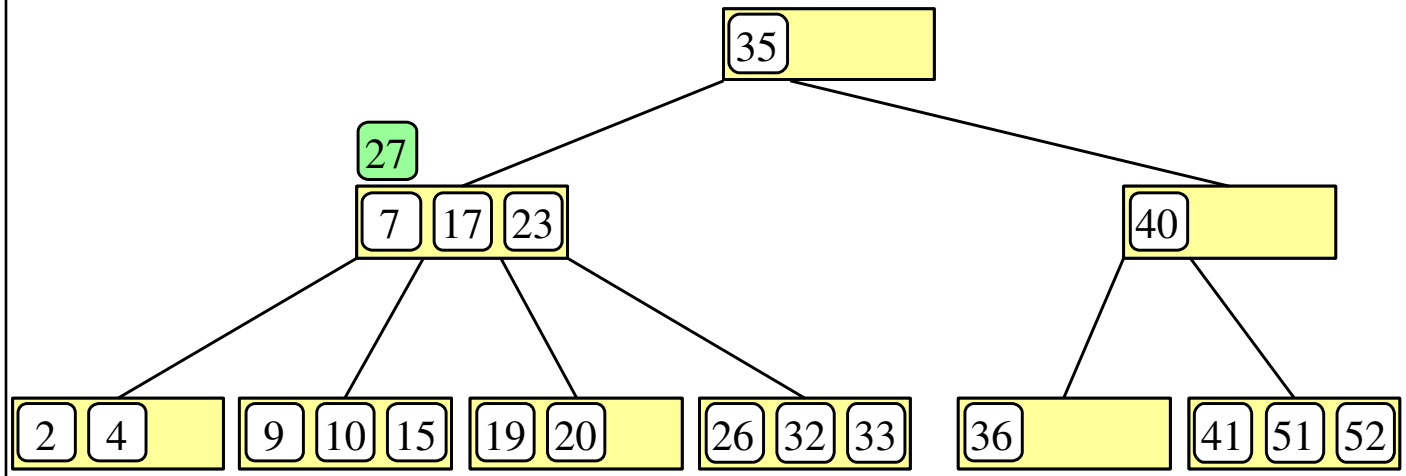
最後に、B-tree への要素の追加について説明します。

この B-tree に 27 を追加することを考えます。

基本的には、member クエリと同様の要素比較をして、木を降りていきます。

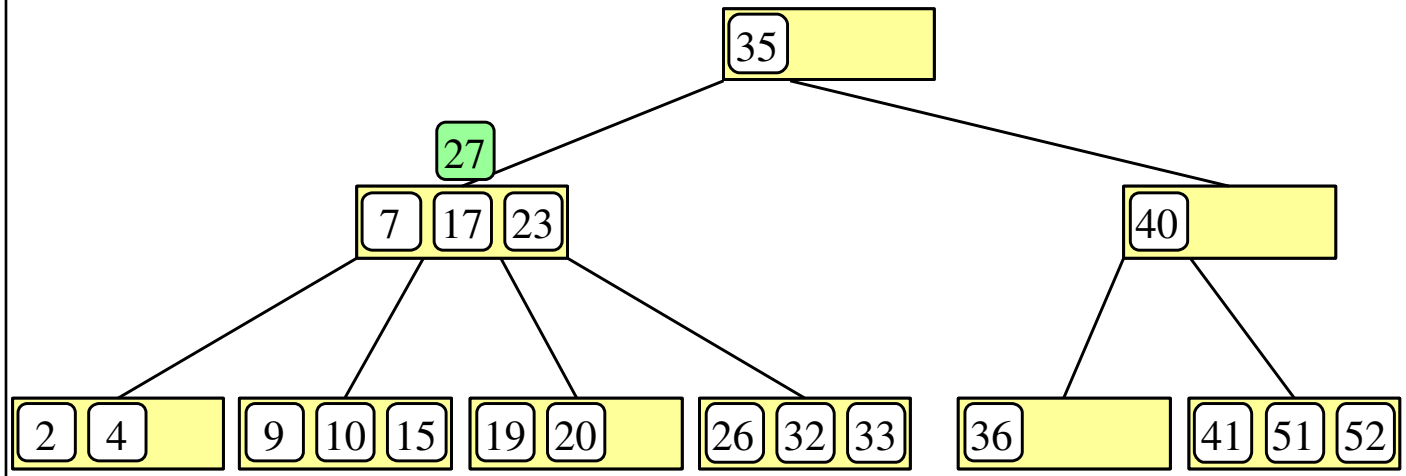
insertion on B-tree

- insert(27, *S*)



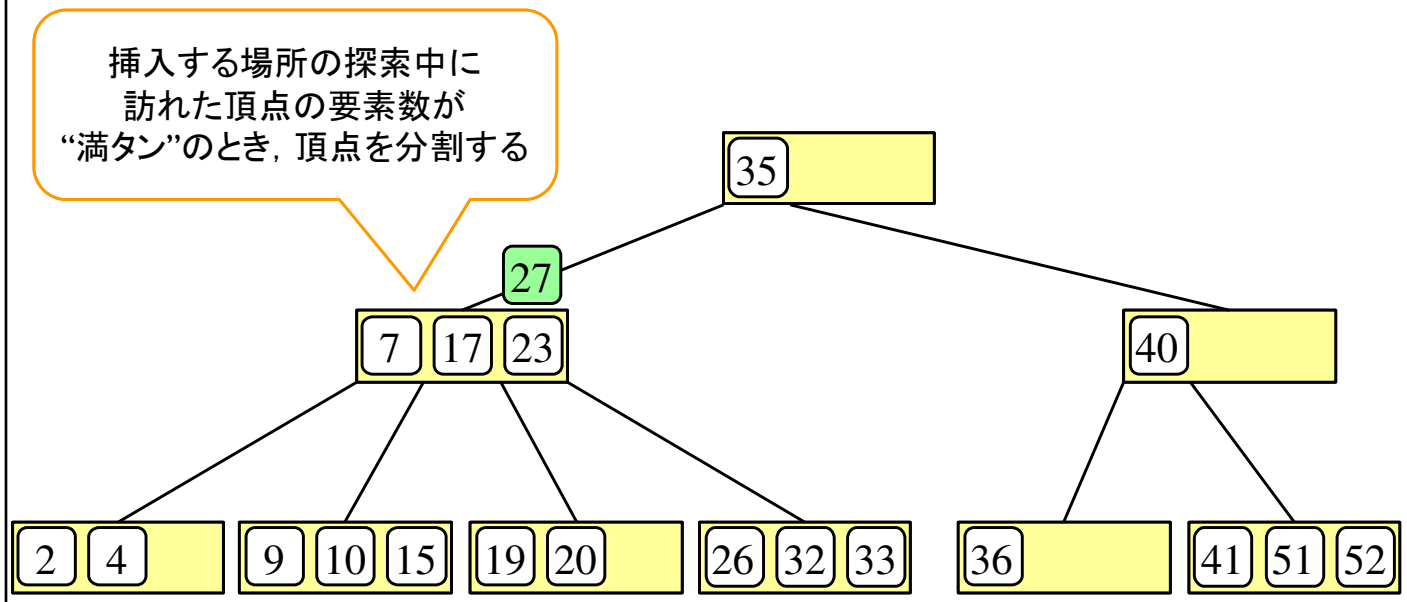
insertion on B-tree

- insert(27, *S*)



insertion on B-tree

- insert(27, *S*)

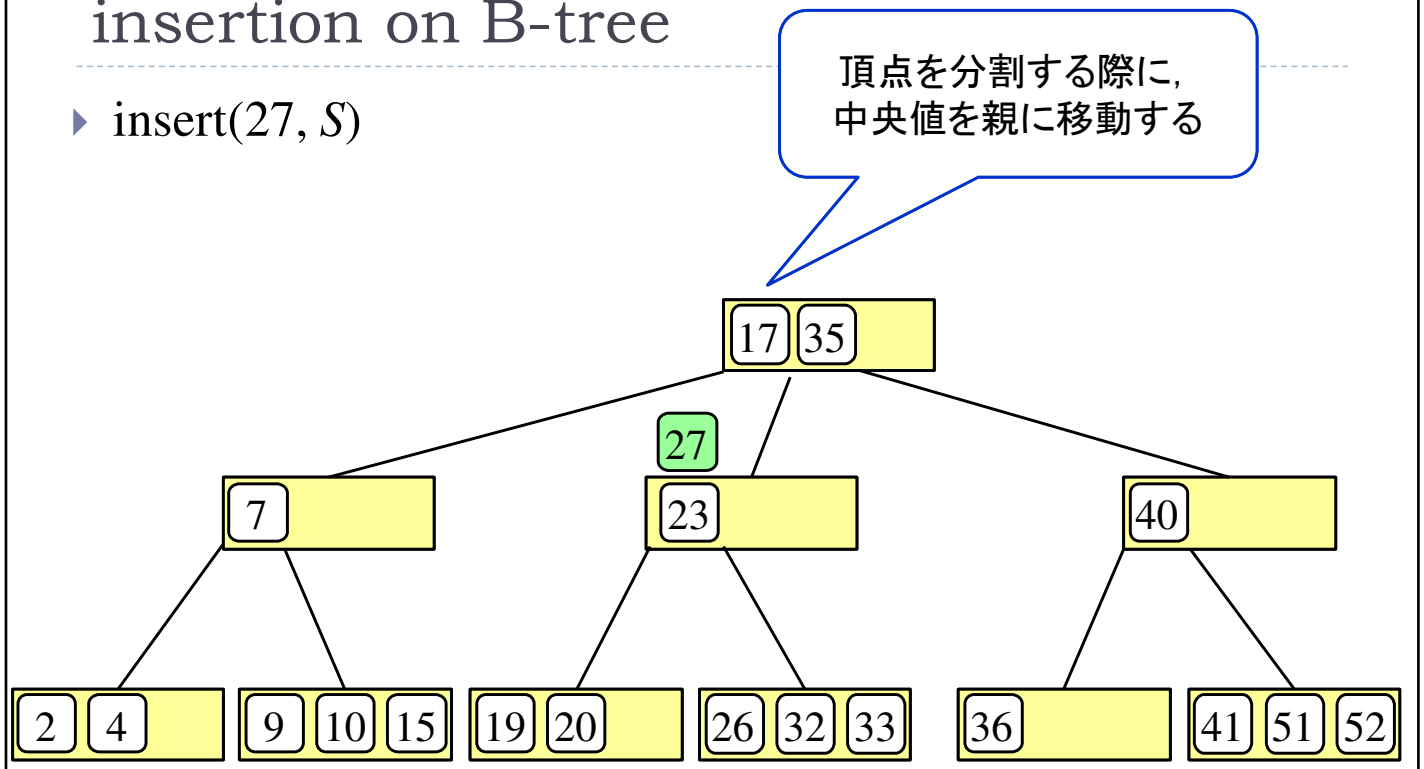


ただし、挿入する場所の探索中に訪れた頂点の要素数が“満タン”のとき、その頂点を分割するという追加作業が発生します。

いま d = 2 なので、各頂点は d-1 = 1 から 2d-1 = 3 個の要素を格納できます。この頂点は 3 個の要素を格納していて”満タン”なので、これを2つに分割します。

insertion on B-tree

- insert(27, *S*)



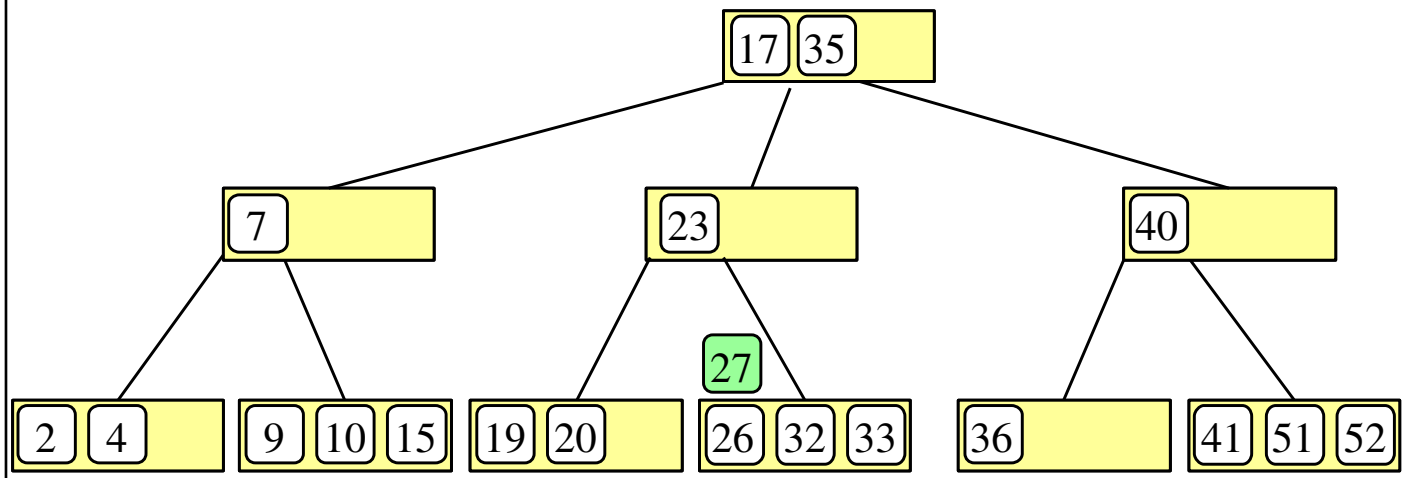
分割後はこのようになります。

分割前の要素を、前半 / 中央値 / 後半 の3つに分割して、前半と後半をそれぞれ分割した頂点に格納します。

一方、中央値は親に移動させます。

insertion on B-tree

- insert(27, *S*)



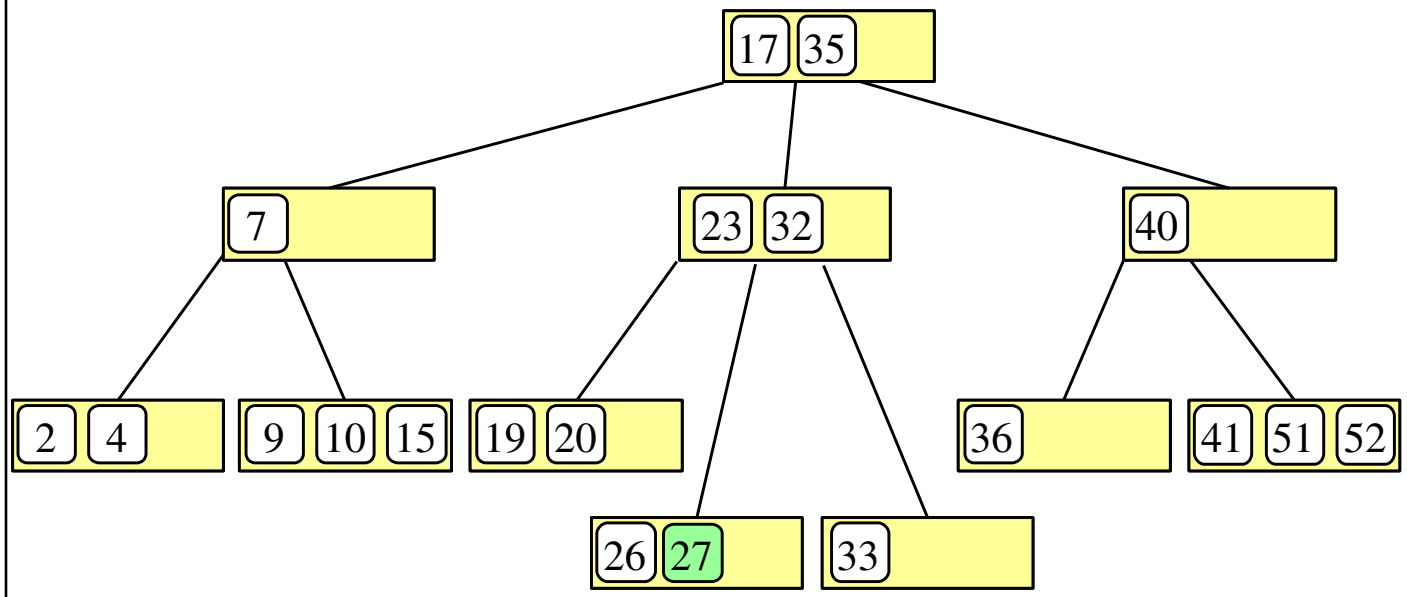
27 の探索を続けます。

27 > 26 なので、この頂点の次の要素 32 と比較します。

このとき、32 > 27 なので、27 はこの 26 と 32 の間に入りたいところですが、この頂点も”満タン”なので、さきほどと同様の分割を行います。

insertion on B-tree

- insert(27, *S*)



分割後の木はこのようになります。これで 27 の追加が完了しました。

ここで、分割前の中央値 32 が親に移動していることに注目しましょう。もし、親も”満タン”だったら、中央値の行先がなくなってしまいます。

しかし、2つ前のスライドで、27 の探索時に通ってきた親をあらかじめ分割しておいたので、32 が入るスペースは必ず確保されています。

こうした工夫によって、この insert のアルゴリズムは正しく要素を追加することができます。

insertion on B-tree

定理 3

B-tree を用いて insert を $O(d \log_d n)$ 時間で実行可能.

【証明】

- 頂点の分割 ➔ $O(d)$ 時間
  - 要素の線形探索 ➔ $O(d)$ 時間
  - 中央値を親に挿入➔ $O(d)$ 時間
- 定理 1 より, B-tree の高さは $O(\log_d n)$.

B-tree を用いて insert 操作を O(d log_d n) 時間で行うことができます。

証明します。

各頂点は高々 2d-1 個の要素を格納しているので、頂点の分割は O(d) 時間で処理できます。

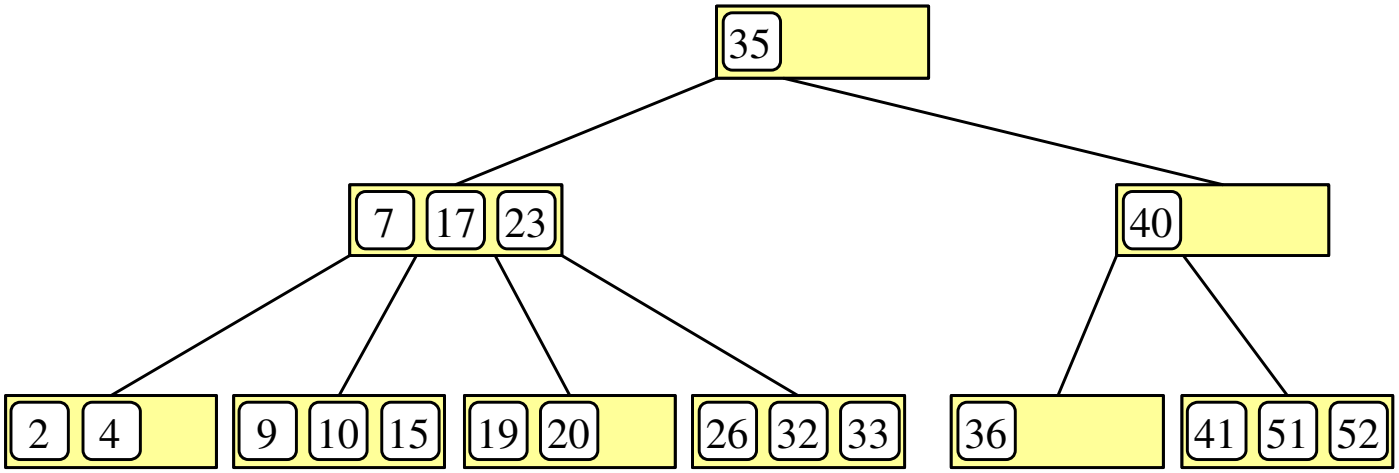
前述の定理 1 より、B-tree の高さは O(d log_d n) です。

よって、これらを掛けることで O(d log_d n) が得られます。

要素の削除についても、同様の計算量で処理することができます。興味のある人は調べてみると良いでしょう。

演習問題 2

- 以下の B-tree に insert(14, *S*) を行って
  得られる B-tree を示せ.



※ 演習問題 **提出〆切：６月４日（木）２３：５９**
pdf の作成方法について、次ページを参照のこと

では、本日の演習問題（その2）です。

この図の B-tree に insert(14, S) を行ったあとに得られる B-tree を図示してください.