システムプログラミング序論
第6回
再帰呼び出し，木構造

大山恵弘

前回の補足

- sizeof(char) は必ず1
  - ISO/IEC 9899:1999 6.5.3.4-3
    “When applied to an operand that has type char, unsigned char, or signed char, (or a qualified version thereof) the result is 1.”

- C++におけるclassとCにおけるstructの違いは小さい

代入式そのもの（代入式全体）の値は，代入された値

- b = (a = 1) + 2;
  → aに1，bに3が代入される
- x = y = 1;
  → x = (y = 1); という意味になり，x にも y にも1が代入される
  - ＝演算子は右結合
- if ((fp = fopen("foo.txt", "r")) == NULL) { ... }
  → fopen の返り値を fp に代入し，さらに，fp と NULL を比較

エラー処理

エラーメッセージの表示に使えるライブラリ関数

- **fprintf**
  - 引数に **stderr** を与えると，指定の文字列を標準エラー出力に出力する

```
fprintf(stderr, “Invalid data: %d¥n”, x);
↓
Invalid data: 10
```

- **perror**
  - 引数の文字列と，最後に実行されたライブラリ関数のエラーメッセージを，標準エラー出力に出力する

```
perror("fopen");
```

perror による出力の例（1）

```
if ((fp = fopen("file0.txt", "r")) == NULL) {
  perror("fopen");
  exit(1);
}
```

```
→ fopen: No such file or directory
```

```
if ((p = malloc(1000000000000)) == NULL) {
  perror("malloc");
  exit(1);
}
```

```
→ malloc: Cannot allocate memory
```

perror による出力の例（2）

```
if ((fp = fopen("file0.txt", "r")) == NULL) {
  perror(NULL);
  exit(1);
}
```

```
→ No such file or directory
```

```
if ((p = malloc(1000000000000)) == NULL) {
  perror("");
  exit(1);
}
```

```
→ Cannot allocate memory
```

プログラムの終了に使えるライブラリ関数

- **`exit`**
  - プログラムを正常終了として終了させ，引数の値を exit status として返す
    - シェルで **`echo $?`** などを実行すると exit status が表示される
    - 使用例： **`exit(1);`**
- **`abort`**
  - プログラムを異常終了として終了させ，異常終了を表すコードを exit status として返す
    - 使用例： **`abort();`**

再帰呼び出し

再帰呼び出し

- 関数内における，その関数自身への呼び出しを，再帰呼び出し（recursive call）と言う
  - 情報科学における重要概念の一つ
  - 色々なアルゴリズムをわかりやすく記述することを可能にする強力な概念

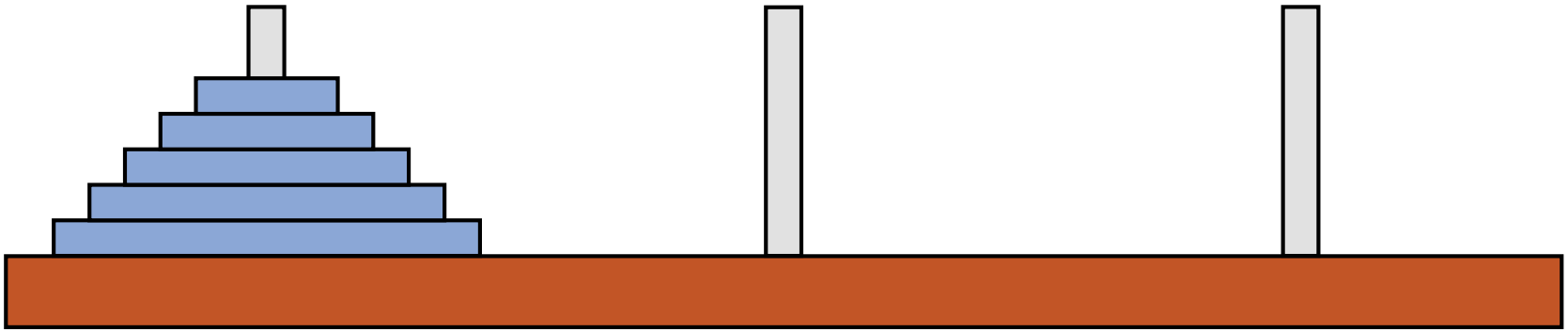
```
int factorial(int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

再帰の考え方

- ある n についての問題を，n - 1 についてのその問題の答を利用して解く
  - より正確に言えば「より簡単に解ける値 m についてのその問題の答を利用して」
  - 例： 99の階乗の値があれば，100の階乗の値を簡単に求められる
  - 例： フィボナッチ数の1000番目と1001番目の値があれば，1002番目の値を簡単に求められる
  - 例： 100個の順不同の整数のリストを昇順にソートしたい場合，もし初めの99個がソートされたリストを得られれば，100個全部がソートされたリストを簡単に得られる
  - 例： 372と108の最大公約数は，108と372 mod 108 (= 48) の最大公約数に等しい

ハノイの塔

- 3本の柱があり，一番左の柱に，下から上に，大きい盤から小さい盤の順に乗っている
- 全部の盤を一番右の柱に移したい，ただし
  - 一度に動かせるのは1枚の盤だけ
  - 小さい盤の上に大きい盤を乗せてはいけない



解法（1）

- まず，2枚の盤だけしかない問題を考えてみる
  - 小さい盤を中央の柱に移し，大きい盤を右の柱に移し，最後に小さい盤を右の柱に移せばよい
- 3枚の盤ではどうか？
  - まず，2枚の盤のときの解法を使って，上2枚を中央の柱に移す
  - 次に，一番下の盤を右の柱に移す
  - 最後に，2枚の盤のときの解法を再び使って，上2枚を中央の柱から右の柱に移す

解法（2）

- 解法を n 枚に一般化する
  - n − 1枚を中央の柱に移す
  - 1番下の盤を右の柱に移す
  - n − 1枚を右の柱に移す
- ある柱から別の柱に n 枚の盤を移す関数を，再帰呼び出しを用いてエレガントに定義できる

数のプリント

- 問題：入力された整数を，1文字の出力関数（putchar）を使って出力する
  - 例えば123という入力に対し，"1"，"2"，"3"の3文字を出力する
- 小さい入力に対する解法
  - 入力 d が1桁の整数の場合
    - 単に d + '0' を出力
  - 入力 d が2桁の整数の場合
    - d / 10 + '0' を出力
    - d を10で割った余りは1桁なので，1桁のときの解法を利用

```
void print_number(int d)
{
  if (d < 10) {
    putchar(d + '0');
  } else {
    print_number(d / 10);
    putchar(d % 10 + '0');
  }
}
```

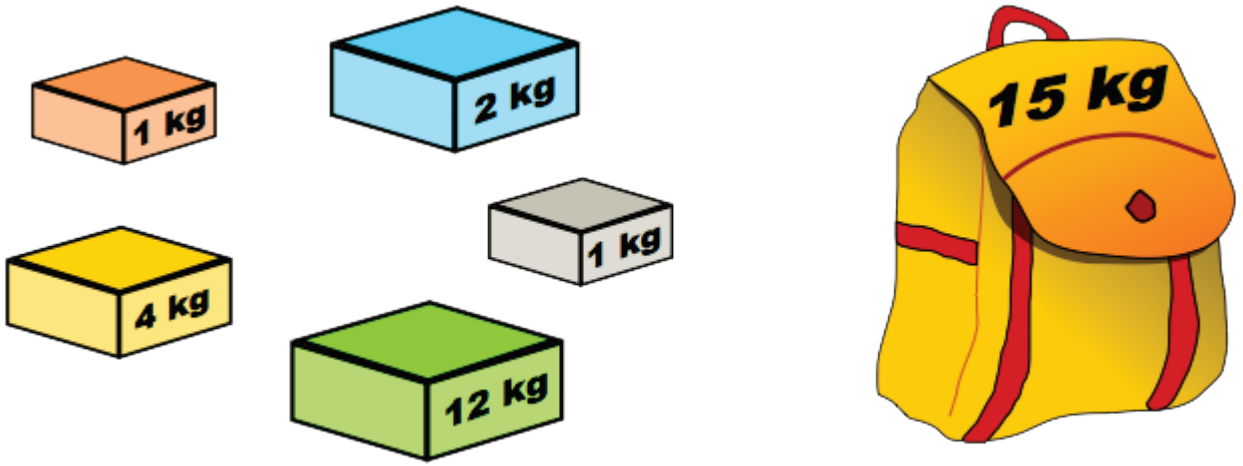
- 一般化解法： 入力 d が n 桁の整数の場合
  - d < 10 ならば，d + '0' を出力
  - そうでないならば，d / 10 （すなわち10の桁まで）をまず表示し，その後，d % 10 + '0' （すなわち1の桁）を出力

別解

```
void print_number(int d)
{
  if (d >= 10) {
    print_number(d / 10);
  }
  putchar(d % 10 + '0');
}
```

ナップサック問題

- 複数ある荷物のうちどれを選んでナップサックに詰めると，荷物の値段の和が最大になるか？
  - N 個の荷物
    - よって，詰め方は 2^N 通り
  - 個々の荷物の重さは W_i，値段は P_i
  - ナップサックに入れられる総重量は W



考え方

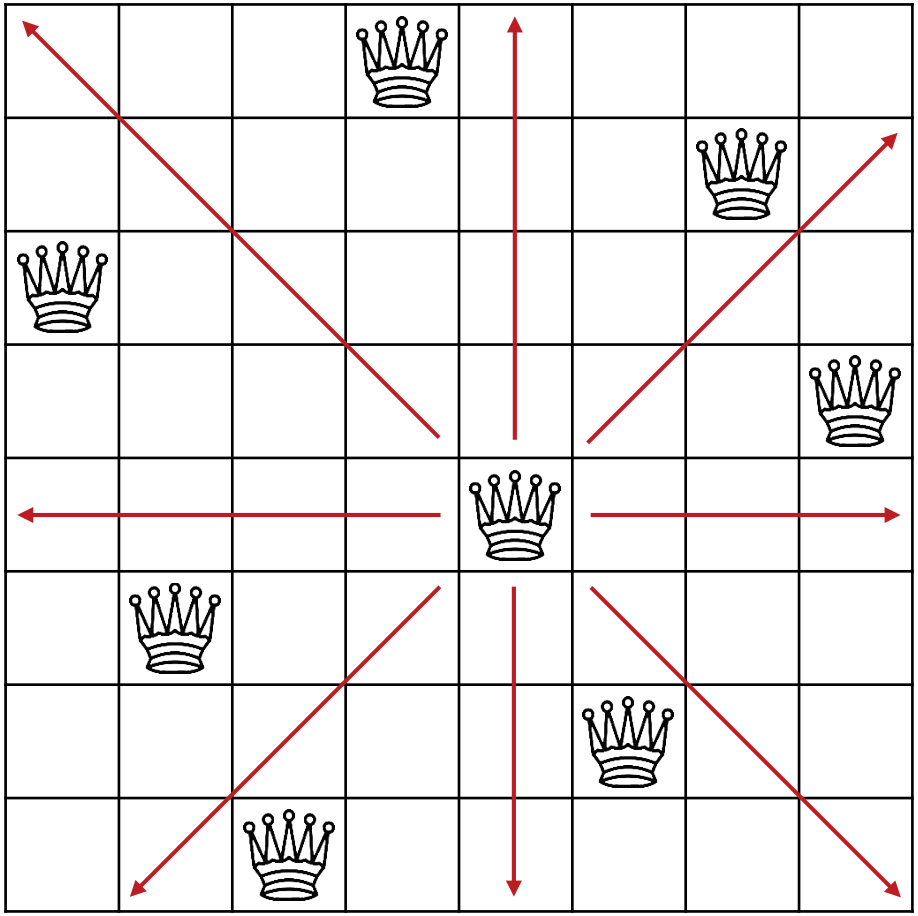
- i 番目の荷物（値段 P_i, 重さ W_i)を入れるかどうか判断する時点で
  - 荷物の値段の和： S
  - さらに詰めることが可能な荷物の重さ： M

  とする
- 入れる場合，S は S + P_i に，M は M - W_i に
- 入れない場合，M と S はそのまま
- 両場合に対して，i + 1 番目以降の荷物をどうするかを判断する

```
int knap_search(int i, int S, int M)
{
  int optl, optr, opt;
  if (i < N && M > 0) {
    if (M >= W[i]) {
      optl = knap_search(i + 1, S + P[i], M - W[i]);
      optr = knap_search(i + 1, S, M);
      if (optl > optr) {
        opt = optl;
      } else {
        opt = optr;
      }
    } else {
      opt = knap_search(i + 1, S, M);
    }
  } else {
    opt = S;
  }
  return opt;
}
```

```
int N;

int main(int argc, char **argv)
{
  int M, P[MAX_N], W[MAX_N];

  データの読み込み...

  opt = knap_search(0, 0, M);
  printf("Highest total price: %d¥n", opt);
  ...
}
```

N クイーン問題



スタックオーバフロー

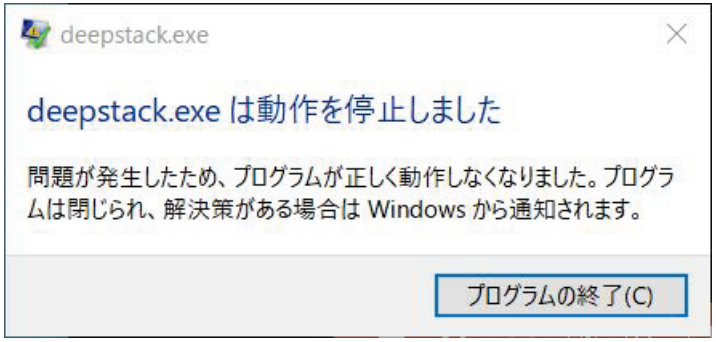
- 再帰呼び出しの段数が増えすぎると，プログラムがメモリ不足で動かなくなることがある

```
#include <stdio.h>

int f(void)
{
  return f() + 1;
}

int main(void)
{
  int x = f();
  printf("%d¥n", x);
  return 0;
}
```

```
crocusxx$ ./a.out
Segmentation fault: 11
```

```
pentas-compc$ ./a.out
セグメンテーション違反です (コアダンプ)
```

```
ubuntu.u.tsukuba.ac.jp$ ./a.out
Segmentation fault (コアダンプ)
```



末尾再帰呼び出し（tail recursive call）

- 関数の実行の最後に行われる再帰呼び出し
  - 再帰呼び出しが返ってきた後に何もせずに，返り値をさらにそのまま return できるような再帰呼び出し
  - 再帰呼び出しで新たにメモリを消費しないことが多い
    - 何段でも再帰呼び出しができる
    - **while** や **for** による繰り返しと同じように扱われる

```
int fact(int n)
{
  if (n == 1) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

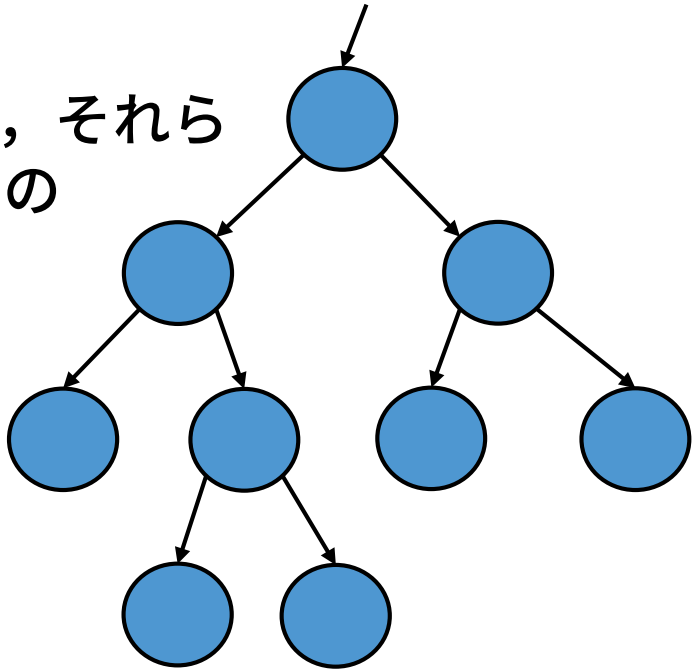
比較せよ

```
int fact_aux(int n, int accum)
{
  if (n == 1) {
    return accum;
  } else {
    return fact_aux(n - 1, n * accum);
  }
}
int fact(int n)
{
  return fact_aux(n, 1);
}
```

木構造

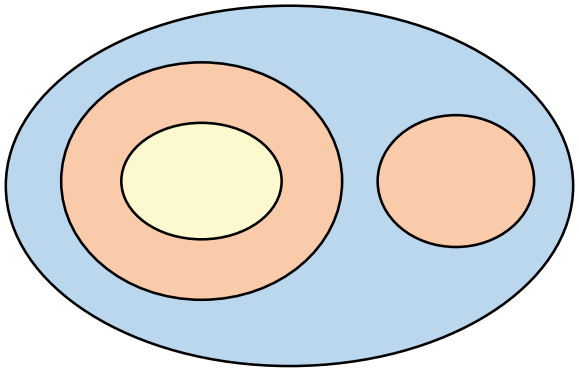
木構造

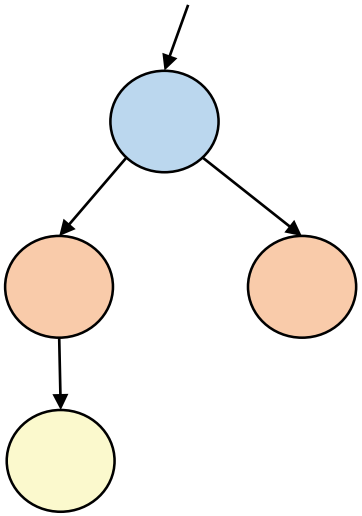
- 接点（node）
  - 値と，他の接点への参照を持つデータ
- 木構造 T
  - 空，または，
  - 木構造 $T_1, T_2, \ldots, T_n$（部分木）と，それらへの参照を持つ接点をあわせたもの
- 接点の種類
  - 根（root）
  - 葉（leaf）
  - 内点（internal node）

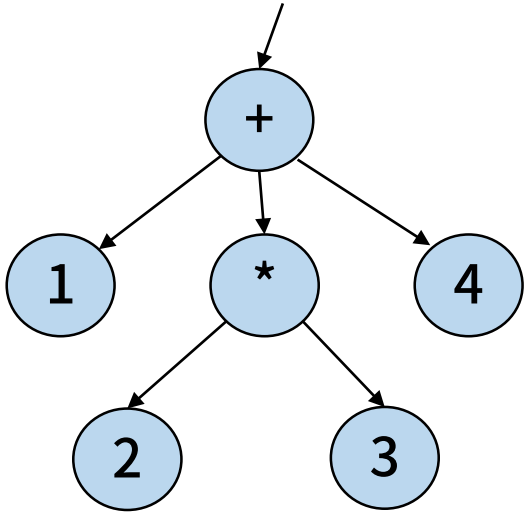


木構造で表されるもの

- 階層的な構造
  - 集合
  - 式
  - …



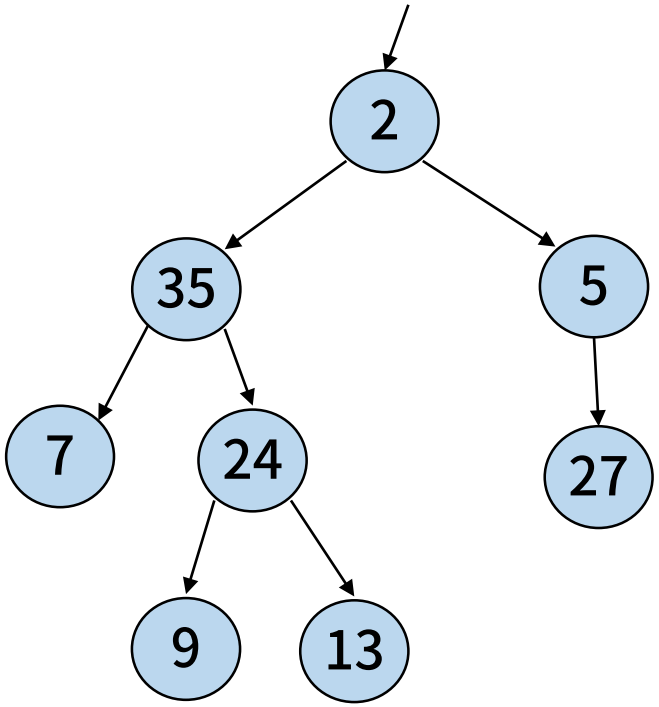




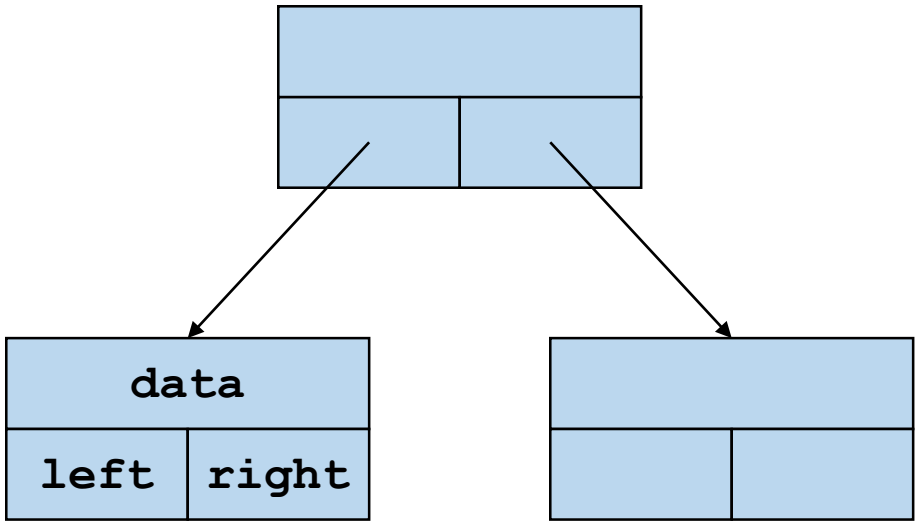
1+2*3+4

二分木

- 2つ以下の部分木への参照を持つ接点のみからなる木

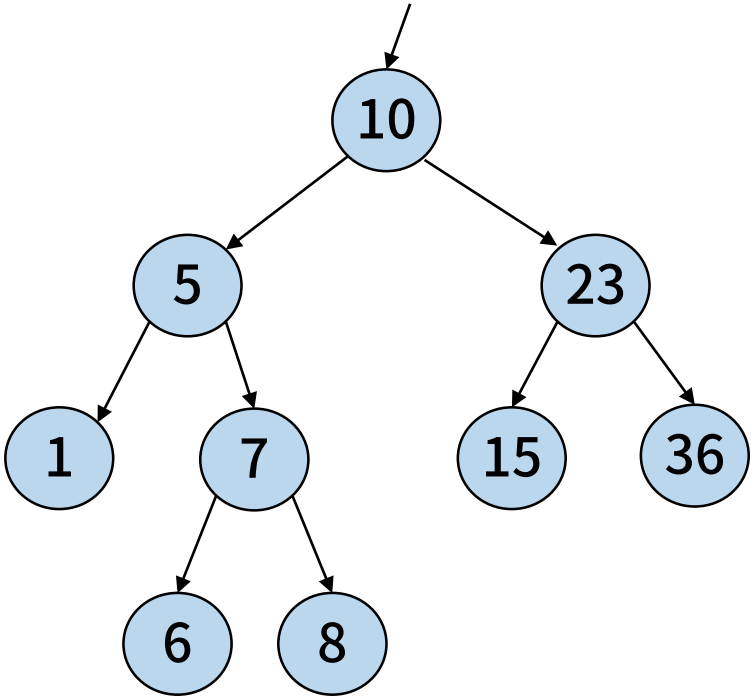


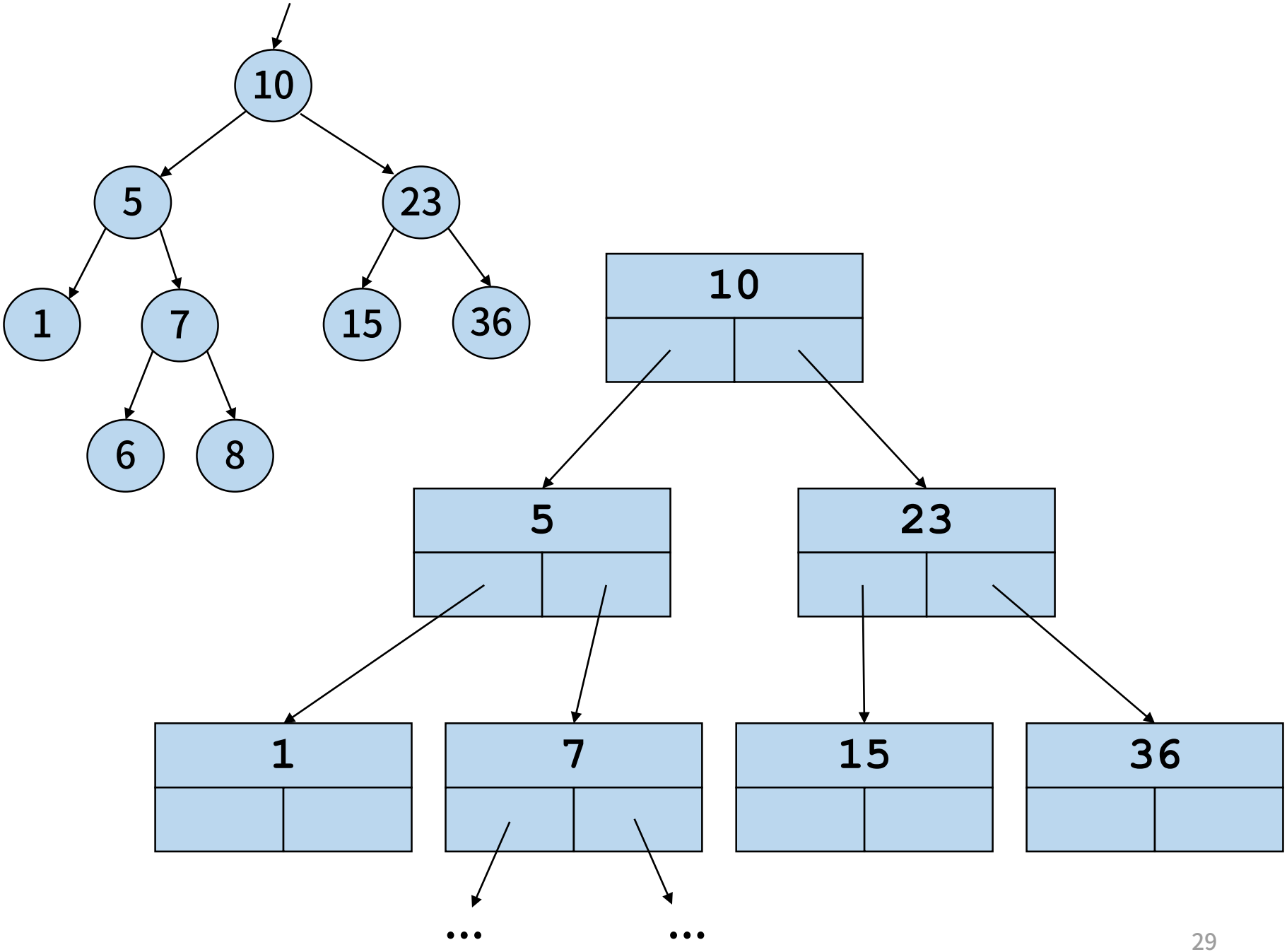

```
struct node {
  int data;
  struct node *left;
  struct node *right;
};
```



ソートされた二分木

- 自分より小さいものは左に，大きいものは右に
- データの高速な探索に適しているので，二分探索木と呼ばれることもある




10
5
23
1
7
15
36
6
8
10
5
23
1
7
15
36
...
...

ソートされた二分木へのデータの挿入

- 考え方（アルゴリズム）
  1. ノードのデータが，入力するデータよりも大きいならば，右の木を対象とする
  2. ノードのデータが，入力するデータよりも小さいならば，左の木を対象とする
  3. もしそこに木が無ければ，ノードを割り当てて，挿入する

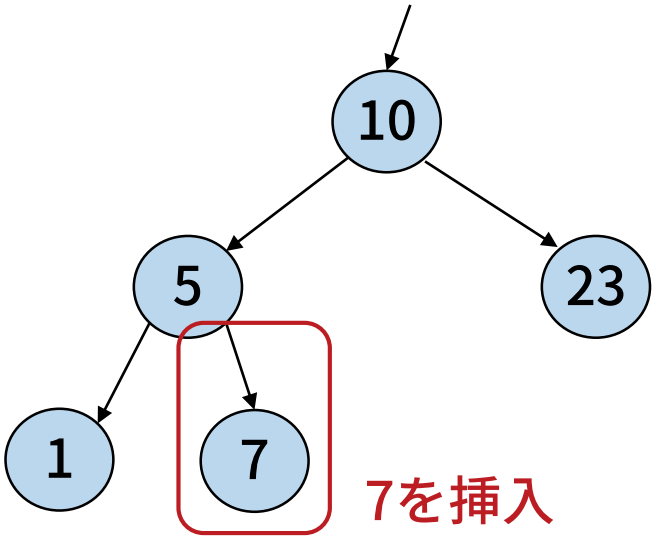
再帰的に考える！

- 2つのバージョン
  - 書き換えられる（かもしれない）ポインタの場所を引数に渡す
    - ポインタを指すポインタを，再帰呼び出しの関数に渡す
  - 挿入後にその場所にあるべき接点へのポインタを返り値として返す
    - 再帰呼び出しの呼び出し側が，返り値を構造体などに格納する

バージョン1

```
void insert_data(char *nm, int pt, struct node **pp)
{
  struct node *p, *nd;
  p = *pp;
  if (p == NULL) {
    nd = (struct node *)malloc(sizeof(struct node));
    if (nd == NULL) {
      fprintf(stderr, "Out of memory¥n");
      exit(1);
    }
    strcpy(nd->name, nm);
    nd->point = pt;
    nd->left = NULL;
    nd->right = NULL;
    *pp = nd;
    return;
  }
  if (pt <= p->point) {
    insert_data(nm, pt, &p->left);
  } else {
    insert_data(nm, pt, &p->right);
  }
}
```

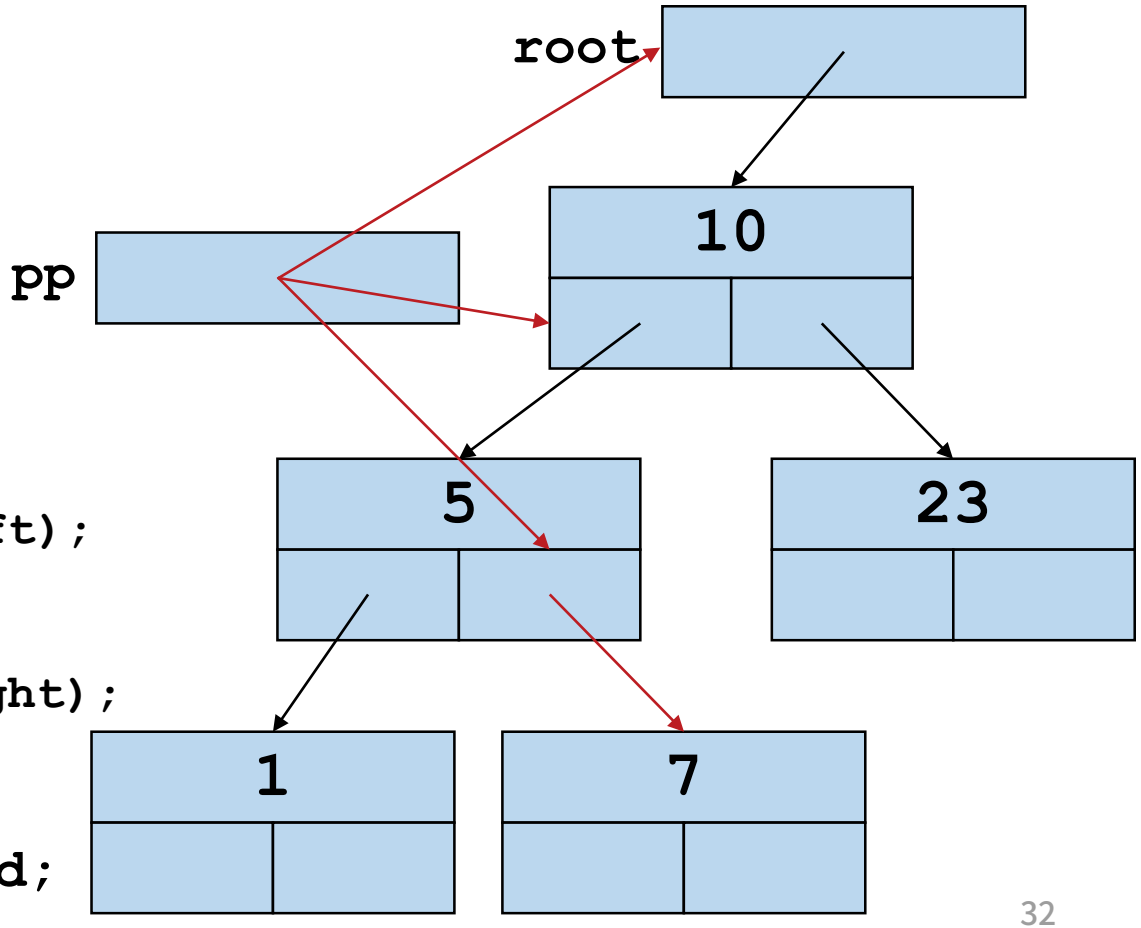
動作




```
if (pt <= p->point) {
     insert_data(nm, pt,
                 &p->left);
   } else {
     insert_data(nm, pt,
                 &p->right);
   }
}
```

```
*pp = nd;
```

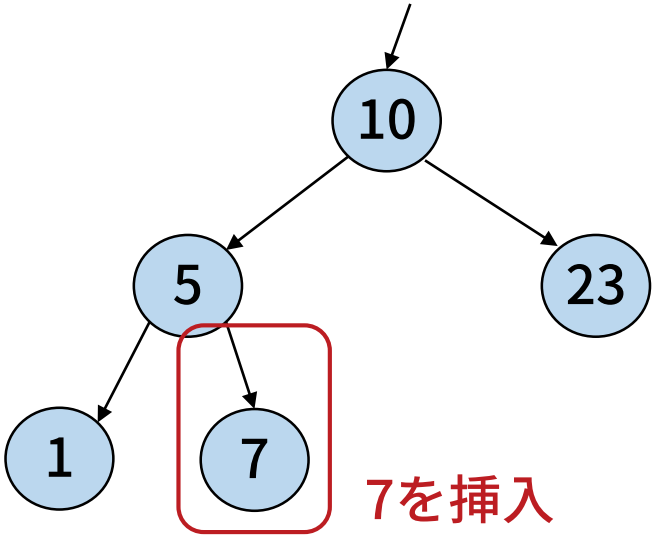
```
void insert_data(char *nm,
  int pt, struct node **pp)
```



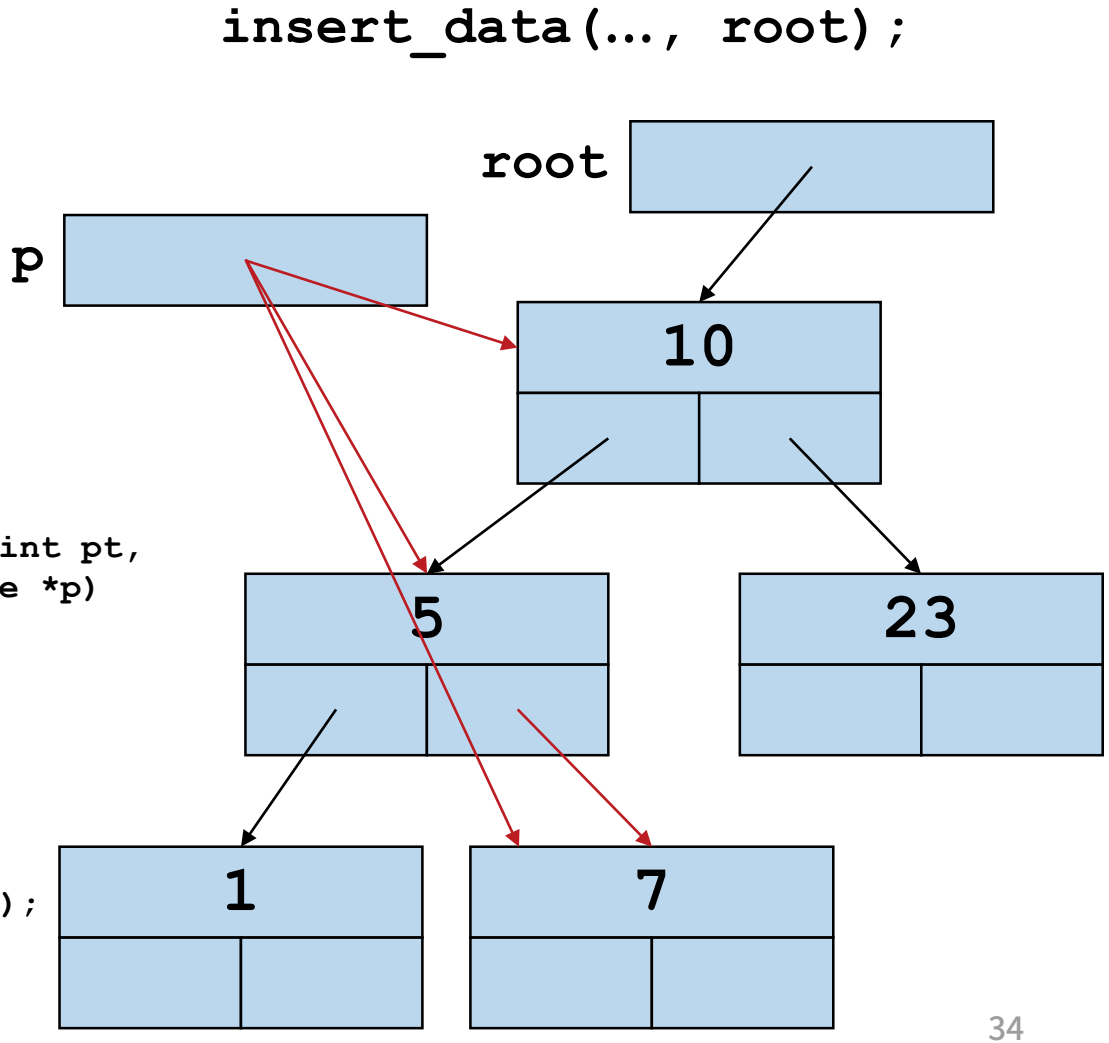
バージョン2

```
struct node *insert_data(char *nm, int pt,
                         struct node *p)
{
  struct node *nd;
  if (p == NULL) {
    nd = (struct node *)malloc(sizeof(struct node));
    if (nd == NULL) {
      fprintf(stderr, "Out of memory¥n");
      exit(1);
    }
    strcpy(nd->name, nm);
    nd->point = pt;
    nd->left = NULL;
    nd->right = NULL;
    return nd;
  }
  if (pt <= p->point) {
    p->left = insert_data(nm, pt, p->left);
  } else {
    p->right = insert_data(nm, pt, p->right);
  }
  return p;
}
```

動作




```
struct node *insert_data(char *nm, int pt,
                         struct node *p)
{
  ...
  if (pt <= p->point) {
    p->left = insert_data(nm, pt,
                          p->left);
  } else {
    p->right = insert_data(nm, pt,
                           p->right);
  }
  return p;
}
```

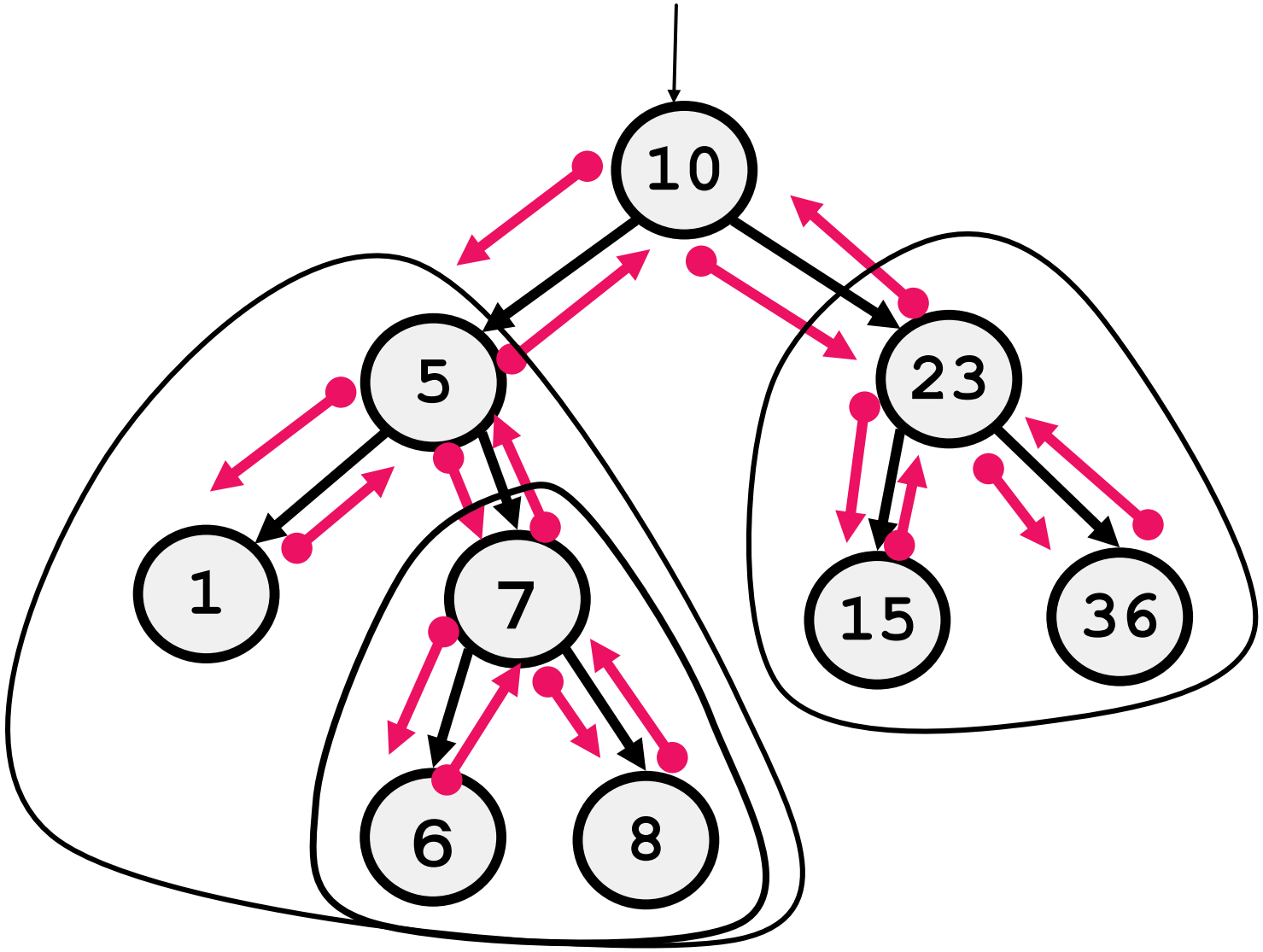


二分木の各接点の情報を，接点が持つ値の昇順に表示

- 左の部分木→現在の接点→右の部分木の順に辿ればよい

```
void print_tree(struct node *p)
{
  if (p == NULL) {
    return;
  }
  print_tree(p->left);
  printf("%s %d¥n", p->name, p->point);
  print_tree(p->right);
}
```

二分木の走査



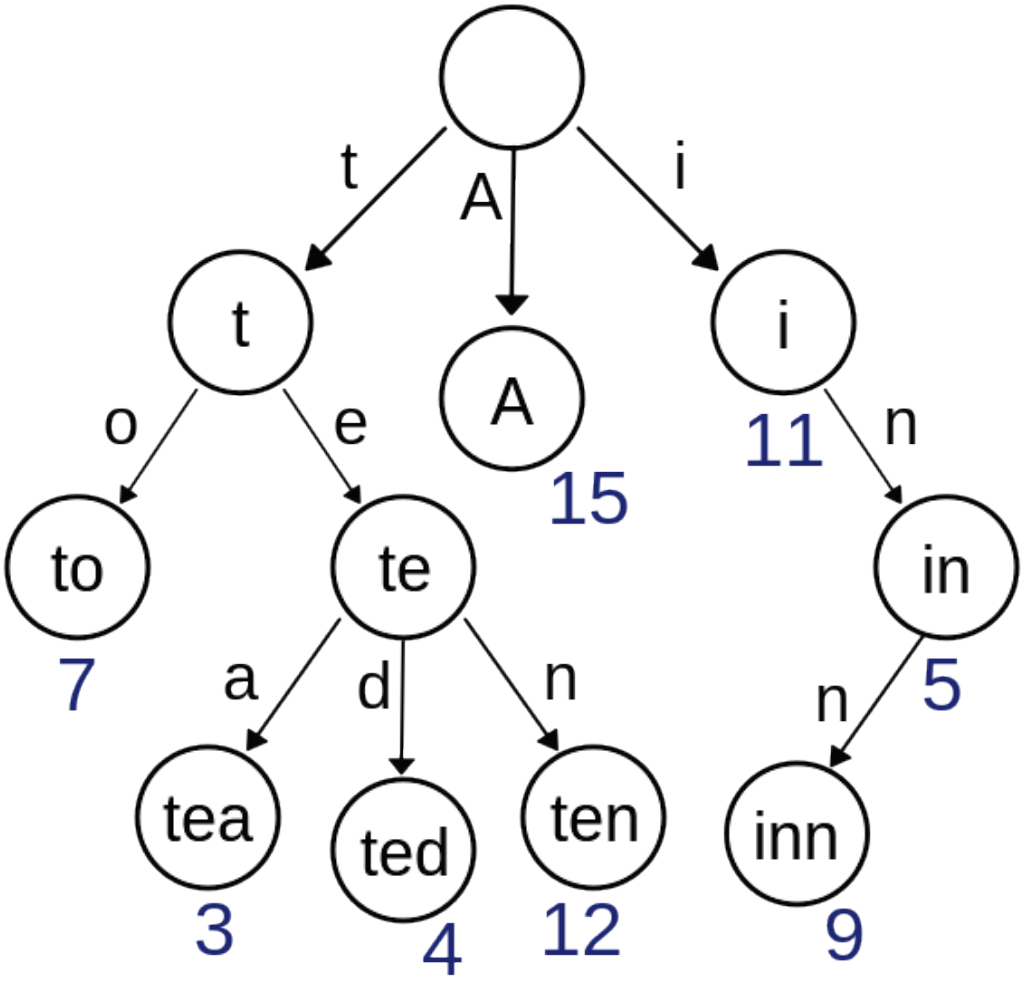
考えてみましょう

- ソートされた二分木から，指定のデータを検索するにはどうするか？
  - データを（ソートされた）リストで管理する方法に比べて，（ソートされた）二分木で管理する方法は，検索が速い
    なぜか？
- ソートされた二分木から，指定のデータを削除するにはどうするか？
- 二分木を消去する（全部の接点のメモリを解放する）にはどうするか？

トライ木

- 木の一種であり，キーとデータの対応を管理する
- キーとして文字列が使われることが多い
- 接点は，キーの接頭部（prefix）を表す
  - キーが文字列の場合，接点は部分文字列を表す
- 木が管理するキーに対応する文字列を表す接点はデータを持つ
  - 全接点がデータを持つわけではない
- 検索が高速であるという利点を持つ

トライ木の例



(Wikipedia より抜粋)

キー	データ
A	15
to	7
tea	3
ted	4
ten	12
i	11
in	5
inn	9

トライ木の操作

- キーの検索
  - "tea", "jet", "idea", "in" に対応するデータがあるかどうかを調べ，あるならそれを返すにはどうするか？
- キー（とデータ）の追加
  - "tedious", "ink", "pen" を加えるにはどうするか？
- キー（とデータ）の削除
  - "tea", "in", "A" を削除するにはどうするか？
- データの更新
  - "ten" のデータを23に更新するにはどうするか？
- キーの列挙
  - すべてのキーを列挙するにはどうするか？

木の種類は多い

- B木（B-tree）
- B+木（B+ tree）
- 赤黒木（red-black tree）
- 平衡二分探索木（self-balanced binary search tree）
- 基数木（radix tree）
- ...

先人の叡智が詰まっている

システムプログラミングで リストや木を取り上げるわけ

- プログラムの低いレベルで何が行われているかを，「手で」理解する
- 動的なメモリの確保と解放，構造体，ポインタの概念を，「手で」理解する
- 低レベルプログラムの随所に出現する，リストや木を管理するコードの定石や型のようなものを覚える

中間試験

- 日時： 2019年12月2日（月） 3限 12:15〜
- 試験時間： 60分
- 場所： 3A202 （この部屋）
- 試験方式： 筆記試験
- 持ち込み不可
- 答案を提出しての途中退出可能
- 試験直後に解答を発表，解説