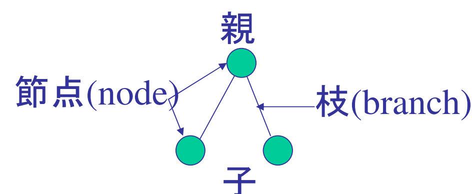
二分探索木によるサーチ

アルゴリズムとデータ構造

- 複雑なプログラムを作成する時，データ構造とアルゴリズムについて考える必要がある
- アルゴリズム
  - 問題を解くための論理または手順をアルゴリズムという
  - ある1問題を解くためのアルゴリズムは，一般に複数個存在する．
  - その中でもより効率よく解答を得られるようなアルゴリズムを見つけることが優れたプログラムを作成する上で大切
- データ構造
  - アルゴリズムを容易にするために工夫されたデータの並び
  - 基本的なデータ構造は，配列，構造体(レコード)型，キュー，スタック，リスト構造，木構造など

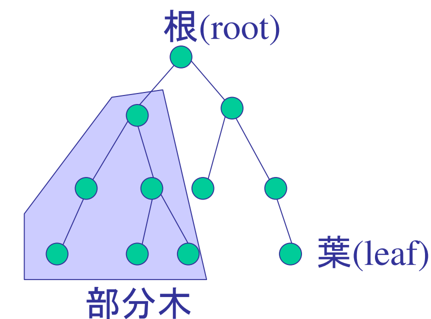
木構造

- 幾つかの節点(node)と，それらを結ぶ枝(branch)から構成
  - 節点がデータに対応
  - 枝がデータ間の親子関係に対応
  - 子：節点の中で下方に分岐する枝の先にあるもの
  - 親：分岐元の節点



図．単純な木構造

木構造



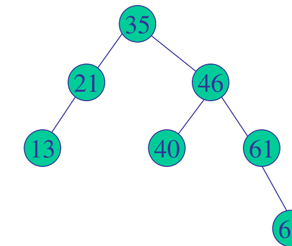
- 根(root)：木の一番上の節点を根(root)
- 葉(leaf)：子を持たない節点
- 部分木：木の中のある節点を相対的な根と考えたときの，そこから枝分かれした枝と節点の集合

二分木(binary tree)

- 木構造で，各節点から出る枝が二本以下のもの
- 木構造に関するアルゴリズムの中で，中心的なデータ構造

二分探索木(binary search tree)

- 二分木の一種
- データの配置に規則あり
  - 左側のすべての子は親より小さい
  - 右側のすべての子は親より大きい
- データの探索のためのデータ構造



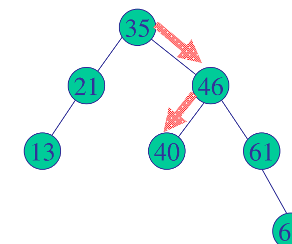
二分探索木による探索

- 根(root)から始める
- 探索キーの値と，各節点のデータを比較し，目標となるデータを探す
  - 探索キーよりも節点のデータが小さいときは，右側の子をたどる
  - 探索キーよりも節点のデータが大きいときは，左側の子をたどる

二分探索木による探索の例

(例) 40である節点を探す場合

1. 根の値(35)と，探索キー(40)を比較
2. 探索キーの方が大きいので，右側の子節点へ移る
3. 次に移った節点の値(46)と探索キー(40)を比較し
4. 探索キーの方が小さいので，左の子節点へ移る
5. 次に移った節点(40)が，目標の節点である

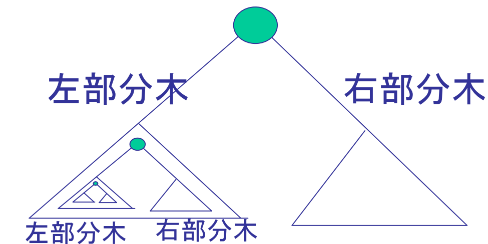


再帰的なデータ構造

- 自分自身を定義するのに，自分自身より1次低い部分集合を用い，さらにその部分集合は，より低次の部分集合を用いて定義するという事を繰り返す構造

二分木の再帰的構造

- 二分木は，各節点の右部分木，左部分木も二分木
- 二分探索木では，探索において，子へ移動し，親での処理と同様の処理を繰り返す



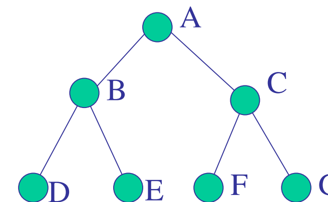
二分探索木の走査

- 走査とは
  - 一定の手順で木の全ての節点を訪れること
- 二分木の節点を巡回する方法
  - pre-order traversal （行きがけ順）
  - post-order traversal（帰りがけ順）
  - in-order traversal（通りがけ順）
- 走査の方法や関数の作り方によって，処理速度が異なる

行きがけ順(pre-order traversal)

- 木の高さを計算する時のような，節上の走査がそれより下の部分木に依存する時に使われる

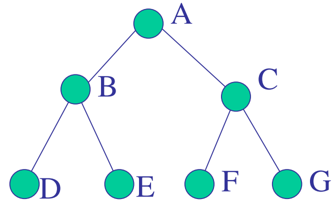
1. 今の節点に操作を施す
2. 左の節点へ移動
3. 右の節点へ移動



A, B, D, E, C, F, G の順に節点を巡回

帰りがけ順(post-order traversal)

1. 左の節点へ移動
2. 右の節点へ移動
3. 今の節点に操作を施す

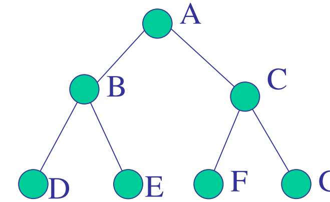


D, E, B, F, G, C, A の順に節点を巡回

通りがけ順(in-order traversal)

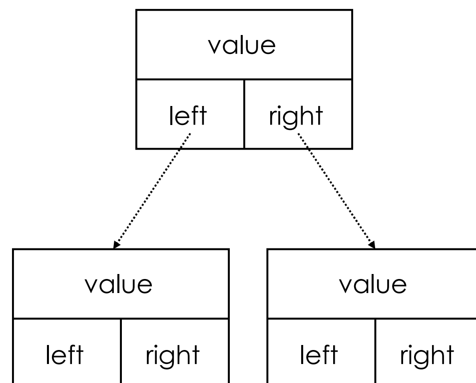
- 二分探索木で，全ての節点のデータを，ソートされた順番で表示するなど

1. 左の節点へ移動
2. 今の節点に操作を施す
3. 右の節点へ移動



D, B, E, A, F, C, G の順に節点を巡回

2分木をC言語の構造体で表現




```
// 2分木のノード
struct  BTNode
{
    BTNode *  left;
    BTNode *  right;
    int           value;
};
```

例題．二分探索木

- 二分探索木の探索プログラムを作る
  - 二分探索木の節点を格納するために，構造体 NODE を作る
  - 二分探索木を格納するために，配列 node を宣言する
    - 行きがけ順に，節点に0から6まで番号を付け，それぞれを node[0], ..., node[6]とする

```
#include<stdio.h>
struct NODE{
    int left;       /*左の部分木*/
    int value;    /*節点の値*/
    int right;     /*右の部分木*/
}
main()
{
    struct NODE nod[] ={{1,35,3}, {2,21,EOF}, {EOF,13,EOF}, {4,46,5}, {EOF,40,EOF},
      {EOF,61,6}, {EOF,69,EOF}};
    int i, in;
    printf("Input Number :");               /*探す値を入力*/
    scanf("%d", &in);
    i=0;
    while(nod[i].value!=in) {
        if ((nod[i].value>in) & (nod[i].left!=EOF)) {      /*左の木を探す*/
            i=nod[i].left;
        } else if ((nod[i].value<in) & (nod[i].right!=EOF)) {   /*右の木を探す*/
            i=nod[i].right;
        } else {
            printf("Not Found¥n");   /*一番下の節点にきても見つからない場合*/
            exit();
        }
    }
    printf("Found %d in nod[%d]¥n", in, i);           /*見つかった場合*/
}
```

末尾再帰最適化

- 末尾再帰を，非再帰的(反復的)なものに変換すること

```
kaijo(int a, int n)
{
   if ( n > 1 ) {
      a = a * n;
      n--;
      return kaijo( a, n );
   }
   return a;
}
int main()
{
   printf( "%d¥n", kaijo(1,5) );
   return 0;
}
```

末尾再帰

```
kaijo(int n)
{
   int f = 1;
   while(n>1) {
      f = f * n;
      n--;
   }
   return f;
}
int main()
{
   printf( "%d¥n", kaijo(5) );
   return 0;
}
```

非再帰

末尾再帰 とは？

- (ある条件の時に)関数の終わりで再帰呼び出しを実行するような関数
- 一般に、再帰関数による処理は、繰り返し処理で書き換え可能である（問題や記述言語によって、どちらの方が書きやすいか異なる）
- ただし、通常は、再帰呼び出しの方が時間的・メモリ的なコストがかかる
- しかし、末尾再帰の場合は、単純に再帰を繰り返しに置き換えられるので、計算機で最適化ができる（特にインタプリタ言語・実行環境で有効）