プログラミング入門 II

複雑なデータ構造とポインタ

佐藤

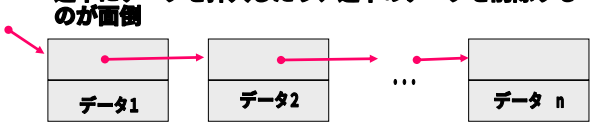
これまでのおさらい（ポインタと構造体）

- ポインタ
  - ポインターってなに？
  - ポインター変数
  - 関数の引数とポインター
  - 配列とポインターの関係
  - データ型
- 構造体
  - 構造体ってなに？
  - データ型と構造体
  - Typedef
  - ポインターを使った構造体の参照
- 動的なメモリ割り当て

リスト構造
＝
構造体
＋
ポインタ
＋
動的な
メモリ割り当て

（線形）リスト構造とは

- 一列に並んだデータをあらわすデータ構造
  - 集合、順序がついたものの並び
  - 必要に応じて、データを確保
- 配列だと
  - データの数の上限をあらかじめ決めておかなくてはならない。
  - 途中にデータを挿入したり、途中のデータを削除するのが面倒



リスト構造の定義

- 自分のデータ型を参照するポインタのメンバーを持つ

```
struct List {
   struct List *next;
   int data;
};
```

ここのデータは場合によっていろいろなものをつけることができる！

リストへのデータの追加

- 大域変数として、リストを持っている変数head
- データを追加する関数addList

```
struct List *head = NULL;

void addList(int x){
  struct List *lp;
  lp = (struct List *)malloc(sizeof(struct List))
  if(lp == NULL){
     printf("no more memory¥n");
     exit(1);
  }
  lp->next = head;
  lp->data = x;
  head = lp;
}
```


リストにあるデータの検索

- リストにデータがあったら、１、なかったら０を返す関数 isExist

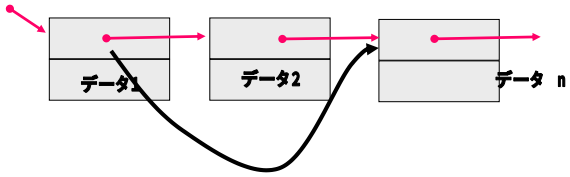
```
void isExist(int x){
  struct List *lp;
  for(lp = head; lp != NULL; lp = lp->next){
    if(lp->data == x) return 1;
  }
  return 0;
}
```

リストにあるデータの削除

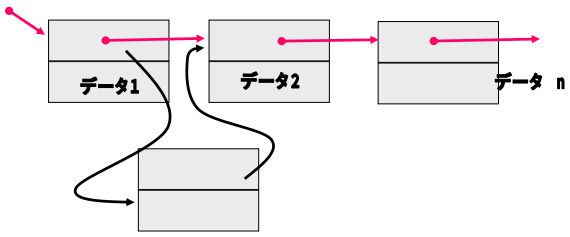
- リストにデータがあったら、削除する removeList

```
void removeList(int x){
  struct List *lp,*lq;
  lq = NULL;
  for(lp = head; lp!= NULL; lp = lp->next){
    if(lp->data == x){
       if(lq == NULL) head = lp->next;
       else lq->next = lp->next;
       free(lp);
       return;
    }
    lq = lp;
  }
}
```


リストからの削除

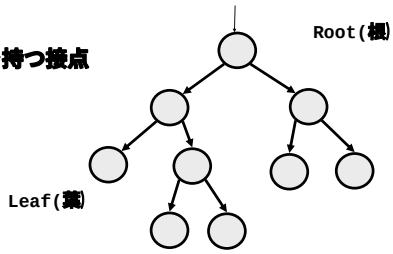


リストへの挿入



木構造

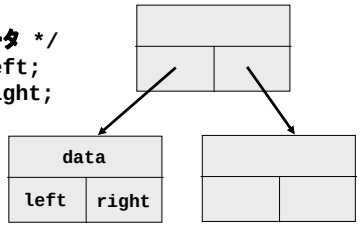
- 木構造Tとは
  - 空
  - Tの有限値の素航増（部分木）をもった型Tの接点
- ２分木
  - ２つの部分木を持つ接点



２分木のデータ構造

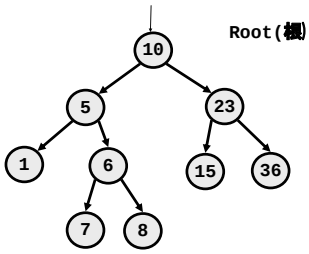
- ２つの部分木left, rightを持つ
  - 線形リストはひとつの部分木（?）を持つ構造

```
struct node {
     /* ノードに持つデータ */
     struct node *left;
     struct node *right;
};
```



木構造を用いたソート

- 小さいものは左に、大きいものは右に



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きいならば、左の木を対象とする
- ③もしも、そこに木がなければ、ノードを割り当てて、挿入する

再帰的に考える！！！

- 実際は、③を最初に行う。

２つのバージョン

- 挿入するところをポインターで渡す
  - ポインターのポインタをつかう
- 挿入したものをポインタで返す
- メインプログラムは、リストのときとほぼ、同じ


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



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


ソート結果の表示

- 左⇒現在のノード⇒右の順でたどればいい。

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


考えてみましょう

- あるデータを検索したい場合はどうすればいいのか？
- 線形リストに比べて、木構造の方が早い！なぜか？
- 木構造からデータを削除する場合はどうすればいいか？