24 PAGES

線形リストによる
スタック・キューの実現

アルゴリズムとデータ構造B

第15回

本日の予定

課題６の確認

- リストの末尾へのノードの挿入，先頭ノードの削除，末尾ノードの削除
- リスト中の全ノードの削除

線形リストの応用

- ノードの探索
- 線形リストによるスタック，キューの実現

問題 1


```
void InsertRear(int x);
```

- リストの末尾にノードを挿入する

- リストの末尾ノードを探索し，そのノードの後ろに新たな末尾ノードを挿入
  - ✓ ノードを動的に生成し，データ部（メンバ `data`）に引数 `x` を代入
  - ✓ ポインタ部（メンバ `next`）は末尾のため `NULL`
  - ✓ **現在の末尾ノードのポインタ部（メンバ `next`）に
新たな末尾ノード（生成したノード）のアドレスを代入**
- 末尾の探索：`struct Element` 型のポインタを用意し，`next` が `NULL` となるまでリストの `head` から順に探索
- 挿入したノードを明示するために課題５で作成した**`Display`**関数を呼び出すこと

- リストが空の場合は，`InsertFront` を呼び出す

問題 1

`void InsertRear(int x);`

- リストの末尾に
- リストの末尾
  - ✓ ノードを動
  - ✓ ポインタ部
  - ✓ **現在の末尾**
    新たな末尾
- 末尾の探索：
- 挿入したノードを明示するために課題3で作成した`Display`関数を呼び出すこと
- リストが空の場合は，`InsertFront` を呼び出す

```
// ノードの挿入は以下で1セット
struct Element *r;
r = malloc(sizeof(struct Element));
r->data = x;  // xは関数の引数

// 末尾ノードの探索
struct Element *p;  // qは不要
for (p=head; p->next != NULL; p=p->next);
// 上のfor文終了後，pが末尾ノードを指している
```

問題 1


```
void RemoveFront();
```

- 先頭ノードを削除する

- 先頭ノードを現在の先頭ノードの次のノードに変更
- 元の先頭ノードの領域を解放
- リストが空の場合は何もしない

- `Remove` 関数の中身にヒントがある

問題 1

`void RemoveFront();`

- 先頭ノードを削除する
- 先頭ノー
- 元の先頭
- リストが空の場合は何もしない

```
// 先頭ノードの削除の基本は以下
head = head->next;

// ただし，先頭ノードの領域をfreeで解放しないといけないため，
// 先頭ノードを別のポインタ（例えば p）で指しておく必要がある
```

- `Remove` 関数の中身にヒントがある

問題 1


```
void RemoveRear();
```

- 末尾ノードを削除する

- リストが空の場合は何もしない
- リストに先頭ノードのみがある場合は `RemoveFront` を呼び出す
- それ以外の場合は，リストの末尾ノードを探索し，末尾ノードの一つ前のノードを新たな末尾ノードに変更し，元の末尾ノードの領域を解放
  - ✓ 末尾の探索に現在のノードをポインタ `p`，一つ前のノードをポインタ `q` が指すループを使用

問題 1


```
void RemoveRear();
```

- 末尾ノ
- リスト
- リスト
- それ以 たな末尾ノー

```
// リストに先頭ノードのみがある場合
// ⇨ 先頭ノードが末尾ノード
if (先頭ノードのnext == 末尾を示すマーク)

// 末尾ノードの探索
struct Element *p, *q;
for (p=q=head; p->next != NULL; q=p, p=p->next);
// 上のfor文終了後，pが末尾ノード，qがその一つ前のノードを指している
// 末尾ノードの領域を解放して，その一つ前のノードを末尾ノードとする
```

✓ 末尾の探索に現在のノードをポインタ p，一つ前のノードをポインタ q が指すループを使用

問題２


```
void Clear();
```

- 線形リスト中の全ノードを削除する

- リストが空になるまで（head が NULL になるまで）
  順に RemoveFront により先頭ノードを削除する
  - ✓ 先頭ノードの値，アドレスを表示して正しいノードを削除（free）しているか確認する

問題２


```
void Clear();
```

- 線形リスト中の全ノードを削除する

- リストが空になるまで（head が NULL になるまで）
  順に RemoveFront により先頭ノードを削除する
  - ✓ 先頭ノードの値，アドレスを表示して正しいノードを削除（free）しているか確認する

```
while (リストが空になるまで) {
    // 先頭ノードのデータ部とポインタ部を表示（確認）
    // RemoveFront関数を呼び出す
}
```

探索

本日の内容はここから

探索

引数：探したい値（キー値）

戻り値：キー値と同じ値を持つノードの `index`
見つからなかった場合は `-1`

- 単純な線形探索で実現
  - ✓ 先頭ノードからから末尾ノードまで順に確認していく
  - ✓ `index` を返すため，ループ回数を保存するループを使用する

- `Remove()` 関数内では値が一致するノードを探して削除を行う
  ⇨ **探索を行っている**

探索

引数：探したい値（キー値）

戻り値：キー値と同じ値を持つノードの index
見つからなかった場合は -1

- 単純な線形探索で実現
  - ✓ 先頭ノー
  - ✓ index
- Remove(
  ⇨ **探索を**

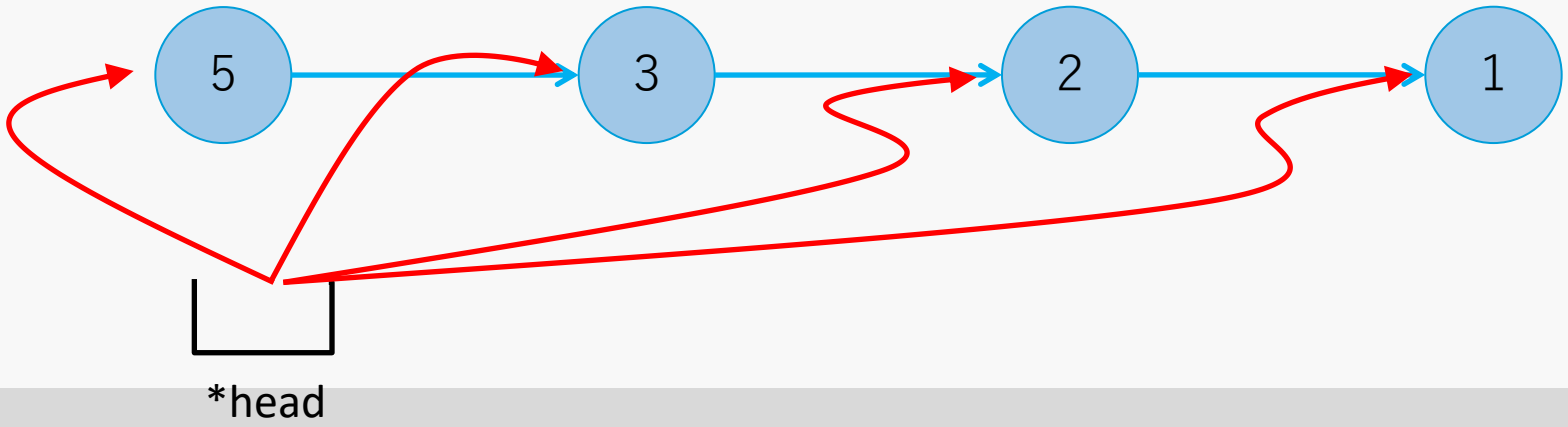
```
int Search (int x) {
    struct Element *p, int i;
    for (p = head, i = 0; p != NULL; p = p->next, i++) {
        if (p->data == x)
            return i;  //indexを返す
    }
    return -1;  //探索失敗
}
```

線形リストによる
スタック，キューの実現

線形リストを利用したスタックの実現

- リストの先頭からのみデータの出し入れを行うことによって、スタックを実現することが可能

- `void Push(int x)` : リストの先頭にデータを入力する

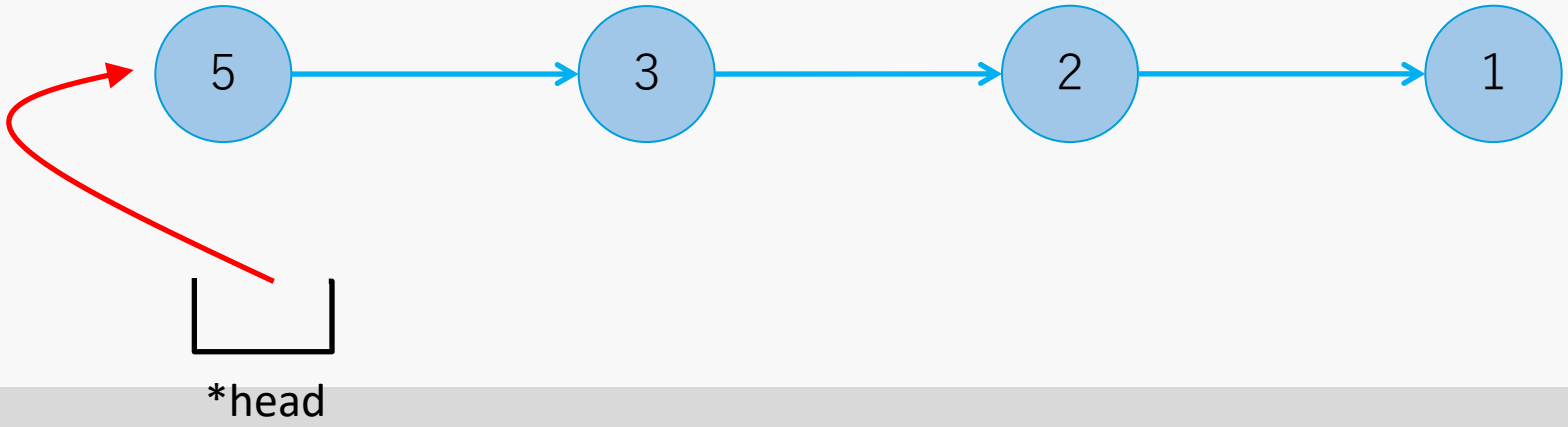


線形リストを利用したスタックの実現

- リストの先
  スタック
- void Pus

```
void Push(int x) {
    struct Element *p;
    p = malloc(sizeof(struct Element));
    p->data = x;
    p->next = head;
    head = p;
}
```

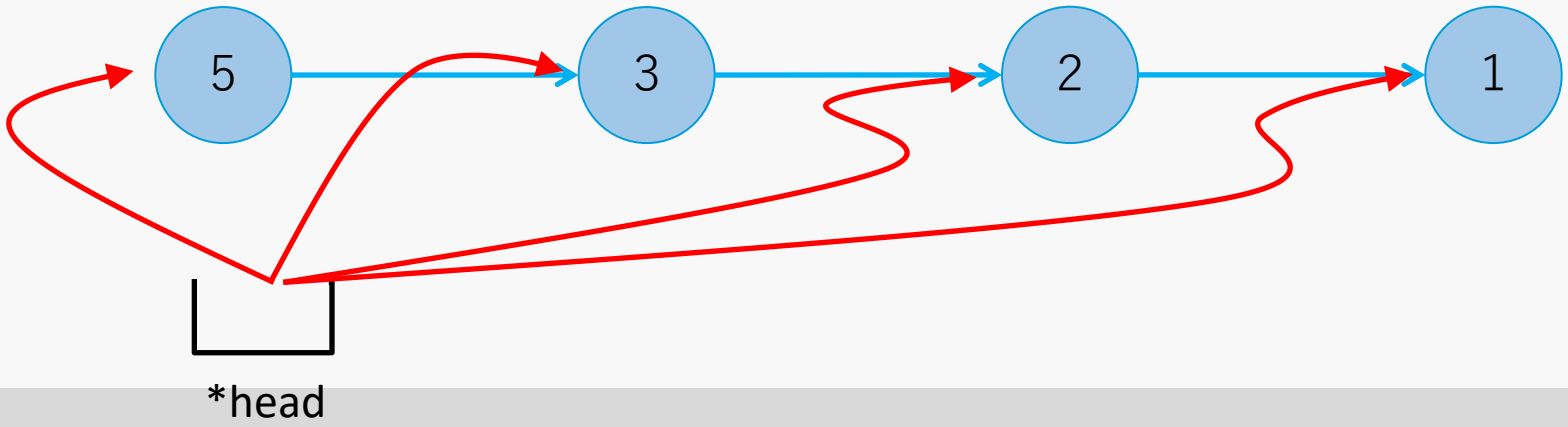
InsertFront() と同一



線形リストを利用したスタックの実現

- リストの先頭からのみデータの出し入れを行うことによって、
  スタックを実現することが可能

- `int Pop()` : リストの先頭からデータを出力する
  スタックが `empty` のときエラー処理として `-1` を返す



線形リストを利用したスタックの実現


```
int Pop() {
    if (head == NULL) // スタックが空
        return -1;
    else {
        struct Element *p;
        int x;
        p =     ;
        x =         ; //先頭データをxに取り出す
        head =          ; //先頭を次のノードに
        free(p); //元の先頭を解放
        return x;
    }
}
```

*head

線形リストを利用したスタックの実現

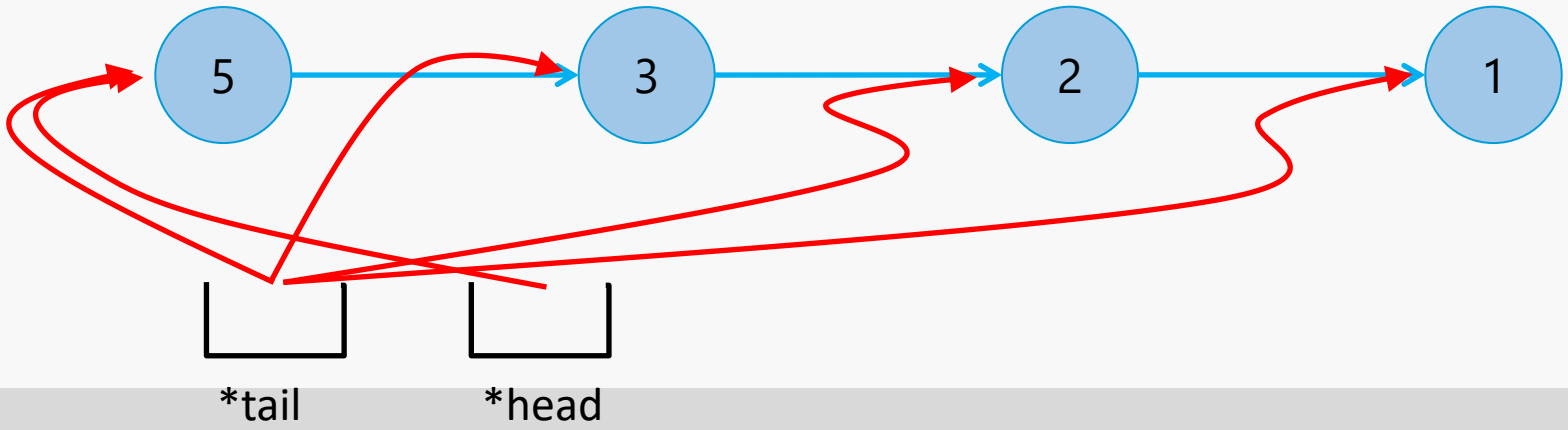
RemoveFront() とほぼ同一

```
int Pop() {
    if (head == NULL) // スタックが空
        return -1;
    else {
        struct Element *p;
        int x;
        p = head;
        x = p->data; //先頭データをxに取り出す（x = head->data; でもOK）
        head = p->next; //先頭を次のノードに（head = head->next; でもOK）
        free(p); //元の先頭を解放（このために p = head; が必要）
        return x;
    }
}
```

*head

線形リストを利用したキューの実現

- データの入力はリストの末尾へのみ，出力は先頭からのみ行うことによって，キューを実現することが可能
- 先頭を指すポインタ `head` に加え，末尾を指すポインタ `tail` を**新たに導入**
  - ✓ 配列で実現した際は：`head` を `front`，`tail` を `rear` と呼んでいた
- `void Enque(int x)` ： リストの末尾にデータを入力する

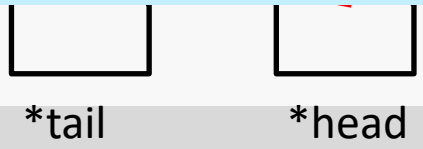


線形リストを利用したキューの実現


```
void Enque (int x) {
    struct Element *p;
    p = malloc(sizeof(struct Element));
    p->data = x;

    if (head ==      ) //キューが空のとき
        head =  ; //新しいノードが先頭となる（※末尾でもある）
    else
                  =  ; //現在の末尾ノードの次が新しいノードとなる

    tail =  ; //新しいノードを末尾ノードにする
    p->next =     ; //末尾ノードのnextをNULLに
}
```

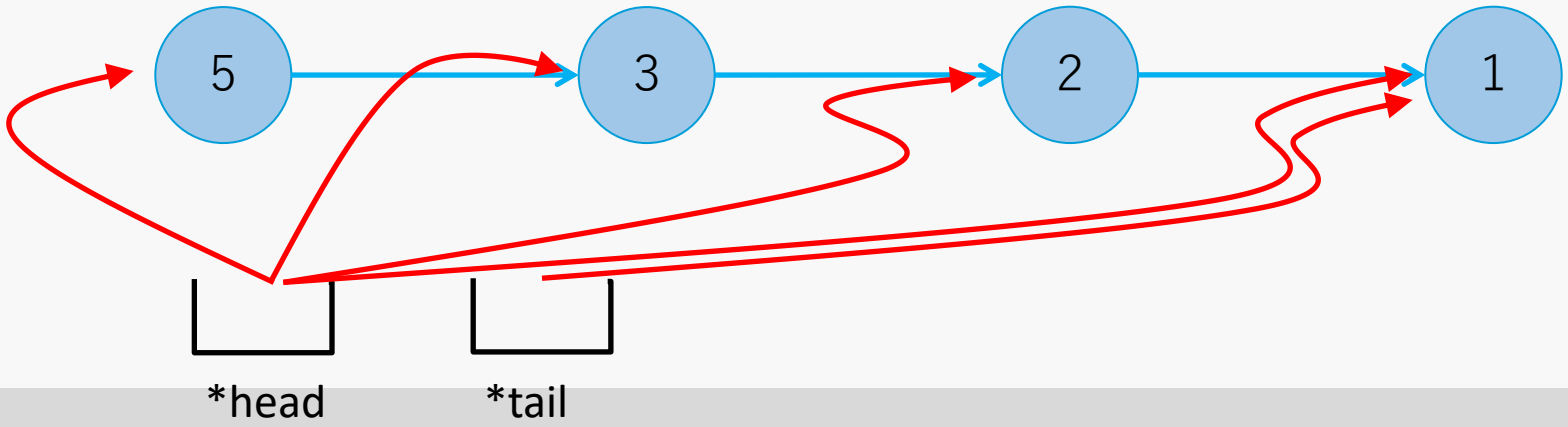
線形リストを利用したキューの実現


```
void Enque (int x) {
    struct Element *p;
    p = malloc(sizeof(struct Element));
    p->data = x;

    if (head == NULL) //キューが空のとき
        head = p; //新しいノードが先頭となる（※末尾でもある）
    else
        tail->next = p; //現在の末尾ノードの次が新しいノードとなる

    tail = p; //新しいノードを末尾ノードにする
    p->next = NULL; //末尾ノードのnextをNULLに（tail->next = NULL; でもOK）
}
```



線形リストを利用したキューの実現

- データの入力はリストの末尾へのみ，出力は先頭からのみ行うことによって，キューを実現することが可能
- 先頭を指すポインタ `head` に加え，末尾を指すポインタ `tail` を**新たに導入**
  - ✓ 配列で実現した際は：`head` を `front`，`tail` を `rear` と呼んでいた
- `int Deque()`: リストの先頭からデータを出力する
  キューが `empty` のときエラー処理として `-1` を返す

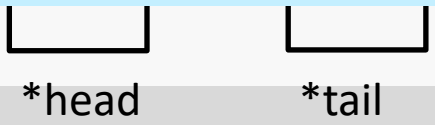


線形リストを利用したキューの実現


```
int Deque() {
    if (head == NULL) // スタックが空
        return -1;
    else {
        struct Element *p;
        int x;
        p = head;
        x = p->data; //先頭データをxに取り出す（x = head->data; でもOK）
        head = p->next; //先頭を次のノードに（head = head->next; でもOK）
        free(p); //元の先頭を解放（このために p = head; が必要）
        return x;
    }
}
```

※ 単方向リストの場合、課題6の RemoveRear() は現実的でない

RemoveFront() とほぼ同一
スタックの pop() と同一



演習

- ex12.c
  - ✓ ノードの探索（Search()関数）を追加で実装しておいてください
  - ✓ 再提出は不要です
- ex15.c
  - ✓ 線形リストによるスタック，キューの実現
    - Push，Pop，Enque，Deque