


データ構造とアルゴリズム 第3回

1. アルゴリズムの重要性
2. 探索問題
-  3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

3

第3章 基本的なデータ構造

- 3.1 配列と連結リスト構造
- 3.2 連結リスト構造の利点
- 3.3 2分探索法に対応するデータ構造
- 3.4 スタックとキューの概念
- 3.5 スタックの実現
- 3.6 キューの実現
- 3.7 ヒープ

4

今日の学習目標

- 基本的なデータ構造を説明できる
 - 配列, 連結リスト, スタック, キュー, ヒープ
- データ構造に対する各種操作とその時間計算量を説明できる
 - 探索, 挿入, 削除

5

3.1 配列と連結リスト構造 (1)

- 配列 (array)
 - 要素は連続したメモリに配置
 - 参照 (k 番目の要素) $O(1)$ 時間 (ランダムアクセス)
 - 列の途中への挿入・削除 $O(n)$ 時間
 - n : 配列内のデータ数
 - 挿入・削除データ以降のデータを1つずつ後・前にずらす
 - 宣言時に大きさを決める
 - 状況によっては困難
 - うまく決めれば, 効率がよい
 - 2次元以上の多次元配列もある

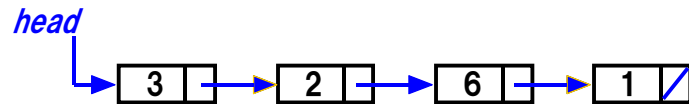
	s
$s[0]$	6
$s[1]$	7
$s[2]$	9
$s[3]$	37
$s[4]$	65
$s[5]$	72
$s[6]$	74
$s[7]$	97

6

3.1 配列と連結リスト構造 (2)

■ 連結リスト

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



第3章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

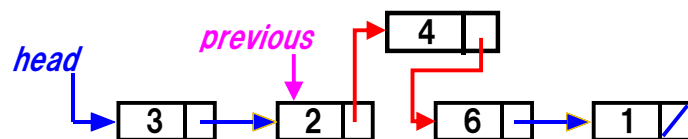
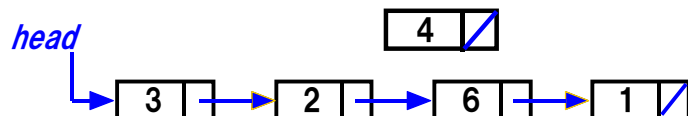
8

3.2 連結リスト構造の利点

●挿入（指定セルの次へ挿入）： $O(1)$ 時間

●最初のセルとしての挿入は特別な処理が必要

- ✓ ポインタheadの書換え
- ✓ 先頭セルをダミー（データを格納しない）とする



リストからの削除

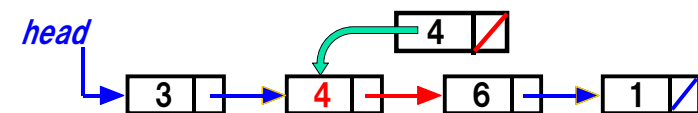
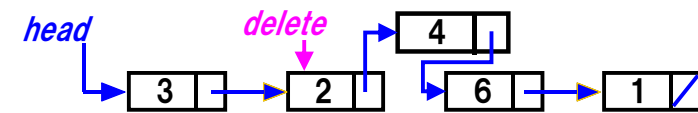
●削除（指定セルの削除）： $O(1)$ 時間

●実際に削除できるのは指定セルの次のセル

- ✓ データコピーによって指定セル削除と等価

●最後のセルの削除は特別な処理が必要

- ✓ 最後のセルの直前のセルを見つける： $O(n)$ 時間
- ✓ 先頭セルをダミー（データを格納しない）とする： $O(1)$ 時間



配列とリストの比較

■ 配列

- 参照 $O(1)$ 時間 (ランダムアクセス)
- 列の途中への挿入・削除 $O(n)$ 時間 (n : データ数)
- データ 1 個あたりの記憶領域: 小
- 宣言時に大きさを決定

■ リスト

- 参照 $O(n)$ 時間 (n : 列の長さ)
- 挿入・削除 $O(1)$
- データ 1 個あたりの記憶領域: 大
- 大きさは動的に変化

11

第 3 章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

12

3.3 2分探索法に対応するデータ構造

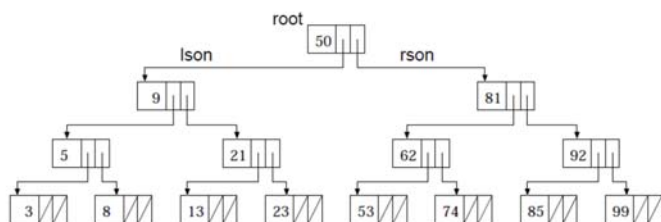
■ 2分探索法

- 探索データとの比較結果で次の比較位置を決定

```
struct BSTnode {  
    int key;  
    struct BSTnode *lson, *rson;  
};
```

lson: 探索キー < 比較キー のときの次の比較位置

rson: 探索キー > 比較キー のときの次の比較位置



13

3.3 2分探索法に対応するデータ構造

■ 探索手続き プログラム 3.5

```
x を入力する;  
v = root (根) とする.  
while (v が NULL でない) {  
    if (x == 節点vのキー値 (v->data)) v を出力して終了  
    if (x < 節点vのキー値) v = v の左の子とする.  
    else v = v の右の子とする  
}  
見つからなかったと報告して終了
```

探索時間: $O(\log n)$

14

第3章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

17

3.4 スタックとキューの概念

■ スタック（プッシュダウンスタック）

■ 有用な抽象データ型

データ構造と操作を定義
実現方法は問わない

- データの列
- 操作：空スタック作成，挿入，削除
- 具体的な実現方法を問わない

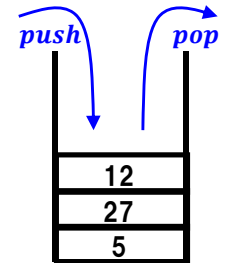
■ データの挿入・削除：列の同じ一方の端だけで行える

■ 操作

- *push*：データの挿入
- *pop*：データの取出し（削除）

■ リストでも配列でも実現可能

- *push*, *pop* とも $O(1)$ 時間



3.4 スタックとキューの概念

■ キュー（待ち行列，FIFOキュー（ファイフオキュー））

■ 有用な抽象データ型

■ データの挿入・削除

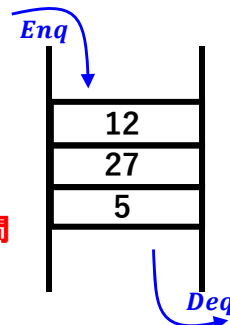
- 挿入を列の一方の端だけ，
削除を列の他方の端だけで行える

■ 操作

- *Enqueue*：データの挿入
- *Dequeue*：データの削除

■ リストでも配列でも実現可能

- *Enqueue*, *Dequeue* とも $O(1)$ 時間



第3章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

20

3.5 スタックの実現

■ 配列によるスタックの実現

```
void push (int x)
{
    stack[top++] = x;
}
```

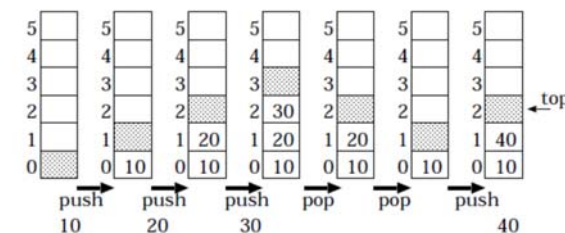
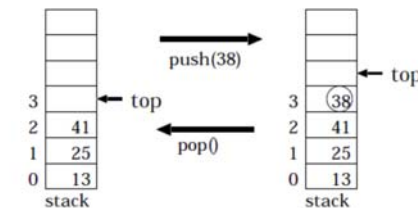
```
int pop (void)
{
    return stack[--top];
}
```

- **オーバーフロー** (*push* でデータ数が配列サイズを超える), **アンダーフロー** (データがないのに *pop*) も考慮する必要あり

21

3.5 スタックの実現

■ 配列によるスタックの実現

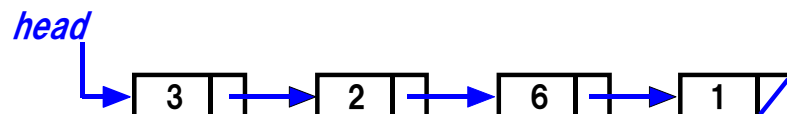


22

3.5 スタックの実現

■ 連結リストによるスタックの実現

- 連結リストの**先頭に挿入** (*push*)
- 連結リストの**先頭から取出し** (*pop*)



23

第3章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

24

3.6 キューの実現

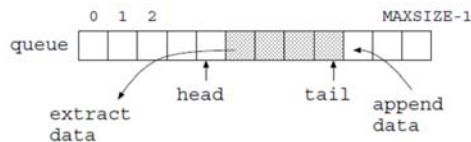
■ 配列によるキューの実現

配列をリングバッファとして利用

```
void Enqueue (int x)
{
    tail = (tail + 1) % MAXSIZE; queue[tail] = x;
}
```

```
int Dequeue (void)
{
    head = (head + 1) % MAXSIZE; return queue[head];
}
```

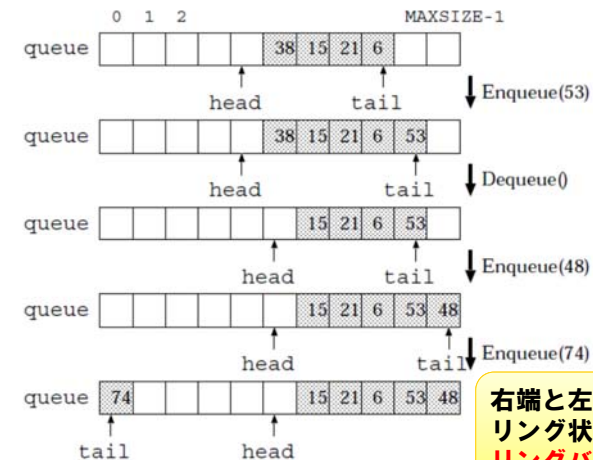
■ オーバーフロー、アンダーフローも考慮する必要あり



25

3.6 キューの実現

■ 配列によるキューの実現



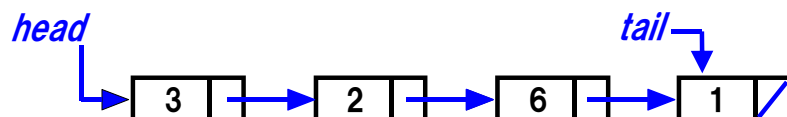
右端と左端がつながった
リング状と見なせるので
リングバッファと呼ばれる

26

3.6 キューの実現

■ 連結リストによるスタックの実現

- 連結リストの最後尾に挿入 (Enqueue)
 - 最後尾を指すポインタ *tail* を用いる
- 連結リストの先頭から取出し (Dequeue)



27

第3章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

30

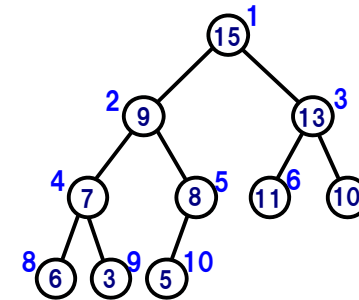
3.7 ヒープ

■ ヒープ

- 2分木を配列 $heap[1..n]$ で実現 (n : データ数)
 - ポインタを使用しない
 - $heap[1]$: 根
 - $heap[k]$ の左の子: $heap[2k]$
 - $heap[k]$ の右の子: $heap[2k + 1]$
- 親のデータ \geq 子のデータ
 - 根は最大のデータを持つ

31

3.7 ヒープ



1	15
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	5

32

3.7 ヒープ

■ データの格納 プログラム 3.13

```
void PushHeap (int x)
{
    int i, j;
    if (++n >= MAXSIZE) stop("Heap Overflow");
    else {
        heap[n] = x;
        i=n; j=i/2;
        while (j>0 && x > heap[j]) {
            heap[i] = heap[j]; i=j; j=j/2;
        }
        heap[i] = x;
    }
}
```

n : データ数

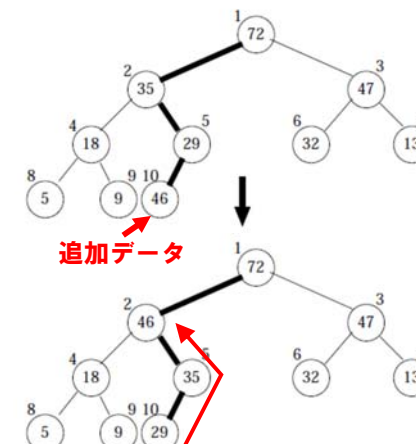
i : データ x の現在ノード
 j : x の現在ノードの親

親ノード j のデータ $< x$
 ノード j のデータを
 ノード i に移動

33

3.7 ヒープ

■ データの格納 プログラム 3.13



34

3.7 ヒープ

■ 最大データの取出し プログラム 3.14

```
int DeleteMax (void) {  
    int x, i, j, t;  
    if (n == 0) stop ("Heap Underflow");  
    else {  
        根のデータが最大  
        x=heap[1]; heap[1]=heap[n--]; i=1;  
        while (i*2<=n) {  
            配列最後のデータを根に移動  
            j=i*2;  
            if (i*2+1<=n && heap[i*2]<heap[i*2+1]) j=i*2+1;  
            if (heap[i]>=heap[j]) break;  
            else {t=heap[i]; heap[i]=heap[j]; heap[j]=t;}  
            i=j;  
            iの子の内、大きいデータを持つ方がj  
        }  
        iとjのデータを交換  
    }  
    return x;  
}
```

根に移動した
データの現在位置

根のデータが最大

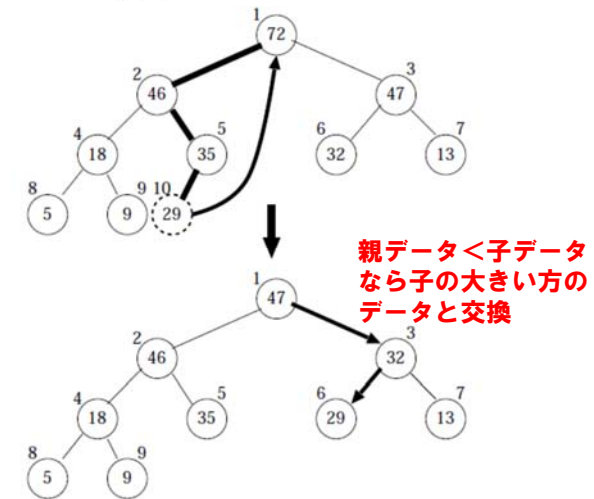
配列最後のデータを根に移動

iの子の内、大きい
データを持つ方がj

iとjのデータを交換

3.7 ヒープ

■ 最大データの取出し プログラム 3.14



36

3.7 ヒープ

■ データの格納

$O(\log n)$ 時間 (n : データ数)

■ 最大データの取出し

$O(\log n)$ 時間 (n : データ数)

まとめ 第3章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 2分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ



今日の学習目標（振返り）

- 基本的なデータ構造を説明できる
 - 配列，連結リスト，スタック，キュー，ヒープ
- データ構造に対する各種操作とその時間計算量を説明できる
 - 探索，挿入，削除