アルゴリズムとデータ構造
第6回: 探索問題に対応するデータ構造(2)

担当: 上原隆平 (uehara)

2015/04/22

内容

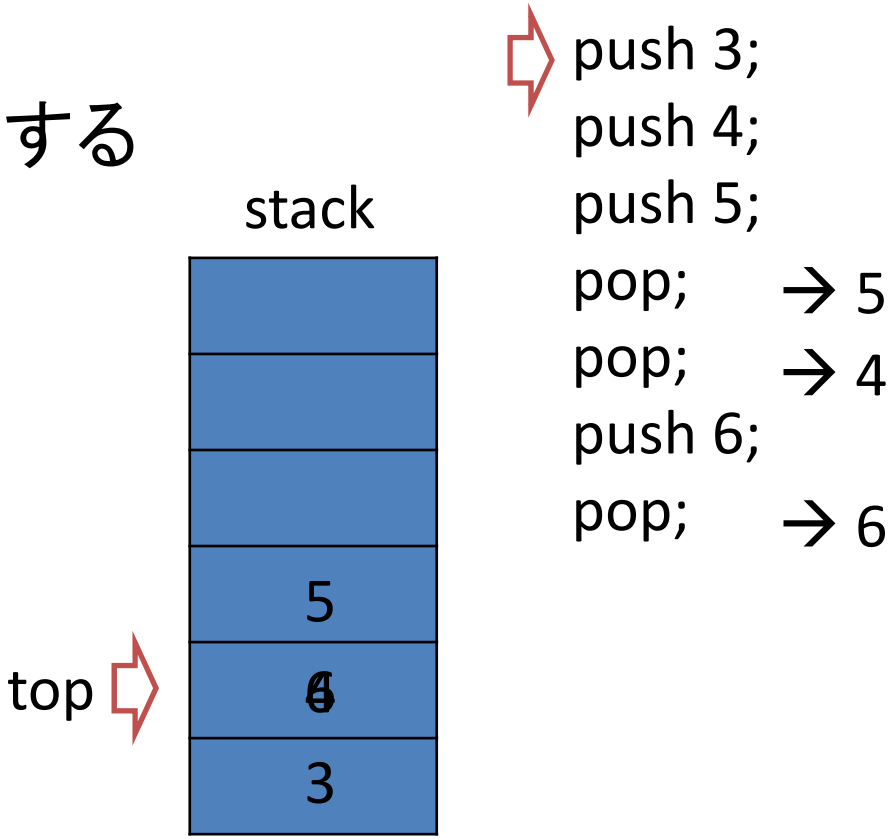
- スタック(stack): 最後に追加されたデータが最初に取り出される
- 待ち行列/キュー(queue): 最初に追加されたデータが最初に取り出される
- ヒープ(heap): 蓄えられたデータのうち小さいものから順に取り出される

- 配列による実装
- 連結リストによる実装

スタック(STACK)

スタック(stack)

- 最後に追加されたデータが最初に取り出される構造(LIFO: Last in, first out)
- 可能な操作
  - push: 新たなデータを追加する
  - pop: データを取り出す
- ポインタ
  - top: stackの先頭
    (次に要素を入れる場所)
    を指す



配列を使ったstackの実装

- データの格納: push(x)

```
stack[top]=x;
top=top+1;
```

- データの取り出し: pop()

```
top=top-1;
return stack[top];
```

- 実行時エラー対策
  - オーバーフロー: top == size(stack)のときにpush(x)
  - アンダーフロー: top == 0のときにpop(x)

配列を使ったstackの実装

```
int stack[MAXSIZE];
int top = 0;
void push(int x){
   if(top < MAXSIZE){
      stack[top] = x; top = top + 1;
   } else
      printf("STACK overflow");
}
int pop(){
   if(top > 0){
      top = top - 1; return stack[top];
   } else
      printf("STACK underflow");
}
```

連結リストによるstackの実装

- 利点: スタックのサイズを宣言する必要無し

```
typedef struct{
  int data; struct list_t *next;
}list_t;
```

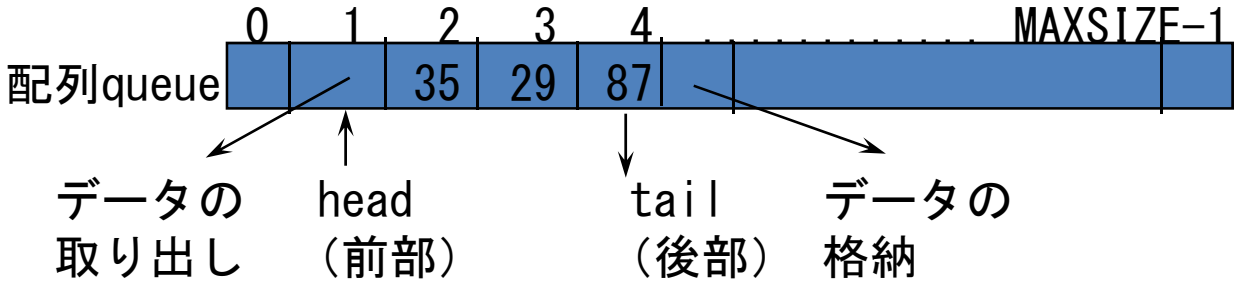
```
list_t* push(list_t *top,int x){
  list_t *ptr;
  ptr=(struct list_t*) malloc(sizeof(list_t));
  ptr->data=x; ptr->next=top; return ptr;
}
list_t* pop(list_t *top){
  list_t *ptr; ptr=top->next; free(top); return ptr;
}
```

ガベージコレクションのある言語では必要無し

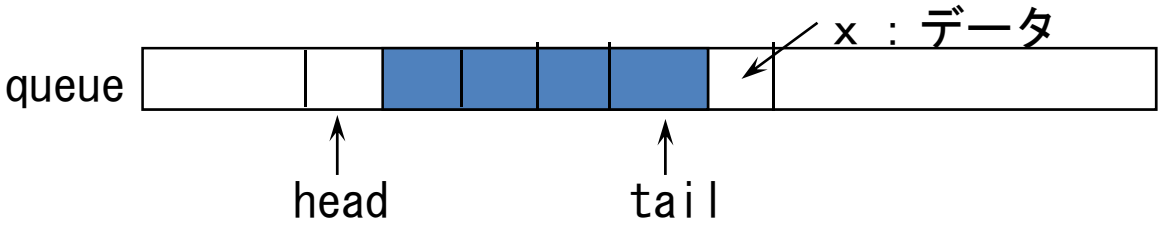
待ち行列/キュー(QUEUE)

待ち行列/キュー(queue)

- 最初に追加されたデータが最初に取り出される(FIFO: first in, first out)

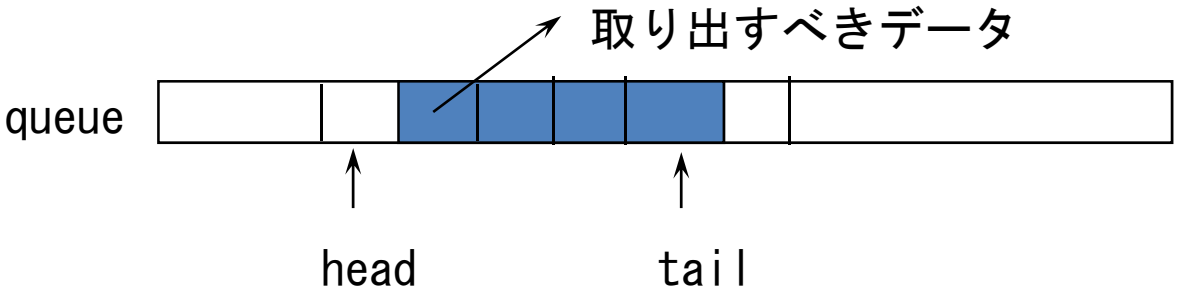


データをqueue[head+1]からqueue[tail]までに蓄える




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

配列によるqueueの単純な実装: データの取り出し




```
int get(){
    head = head + 1;
    return queue[head];
}
```

単純なqueue実装の問題: 使っているうちに無駄ができる

- 以下のようにqueueを使うと、配列はどう使われる?

```
int queue[MAX_SIZE];
int head, tail;
void main(){
  head=0; tail=0;
  append(3); get();
  append(4); get();
}
```

```
int get(){
  head = head + 1;
  return queue[head];
}
```

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

解決: 配列を環状に使う

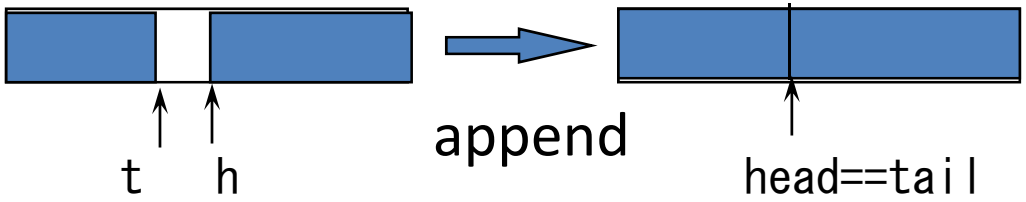

```
void append(int x){
  tail = (tail + 1) % MAXSIZE;
  queue[tail] = x;
}
int get(){
  head = (head + 1) % MAXSIZE;
  return queue[head];
}
```

端まで行ったら0に戻る

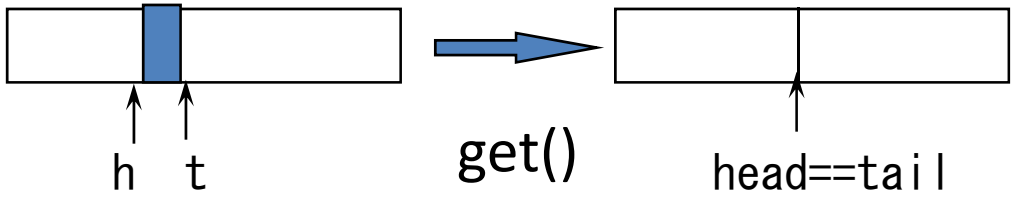
端まで行ったら0に戻る

環状利用の問題:
満(full)と空(empty)が区別不能

fullのとき



emptyのとき:



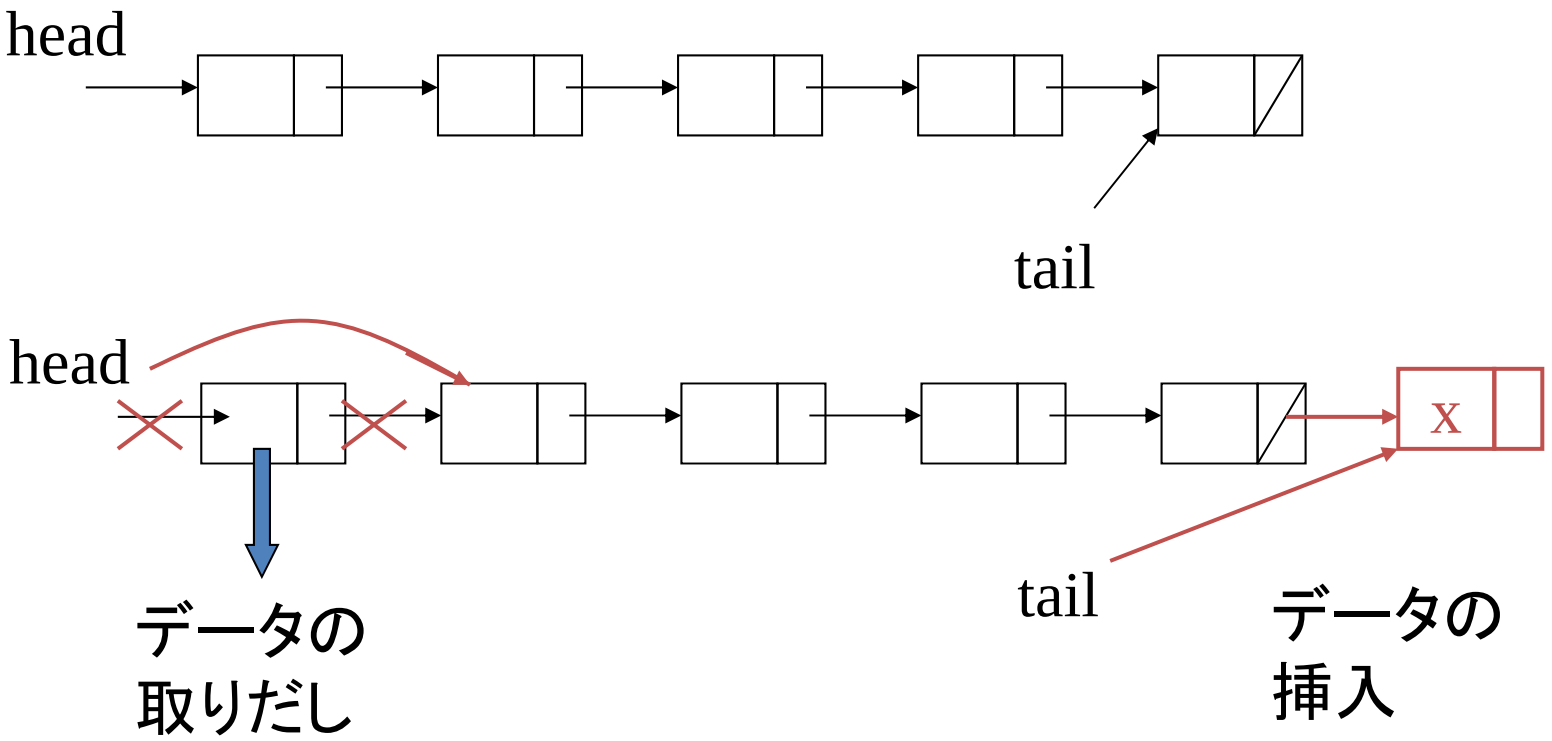
どちらの場合もhead==tailで区別できない

解決: append時にtail==headとなった時をfullとする

```
void append(int x){
  tail = (tail + 1) % MAXSIZE;
  queue[tail] = x;
  if(tail == head) printf("Queue Overflow ");
}
int get(int x){
  if(tail == head) printf("Queue is empty ");
  else {
    head = (head + 1) % MAXSIZE;
    return queue[head];
  }
}
```

連結リストによるqueueの実現

データの挿入:リストの末尾から　　tailポインタ
データの取り出し:リストの先頭から　　headポインタ



プログラムは練習問題とする

- 配列を用いた単純な実装
- 2分木の考え方を用いた配列による実装

ヒープ(HEAP)

ヒープ(heap)

- データの格納ができる
- 最小(または最大)の要素から順に取り出される

Q. どうやって実現する?

ヒープの実現(1):
配列を使った単純な

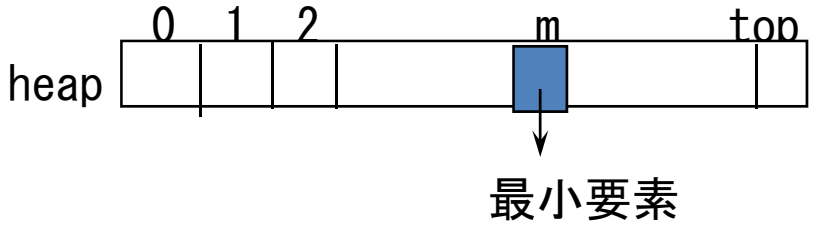
1次元配列heap[]とデータ数を表す変数topを用意

- 初期設定: top = 0
- データの格納:

```
heap[top] = x;
top = top + 1;
```

- 最小要素の取り出し:
  配列heap[]の中で最小の要素heap[m]を求め、これを出力した後、右端の要素heap[top-1]をheap[m]の位置に移し、topの値を1減らす

```
m = 0;
for(i=1; i<top; i++)
  if(heap[i] < heap[m])
    m = i;
x = heap[m];
heap[m] = heap[top-1];
top = top - 1;
return x;
```



配列による単純な実現の問題: データの取り出しが遅い

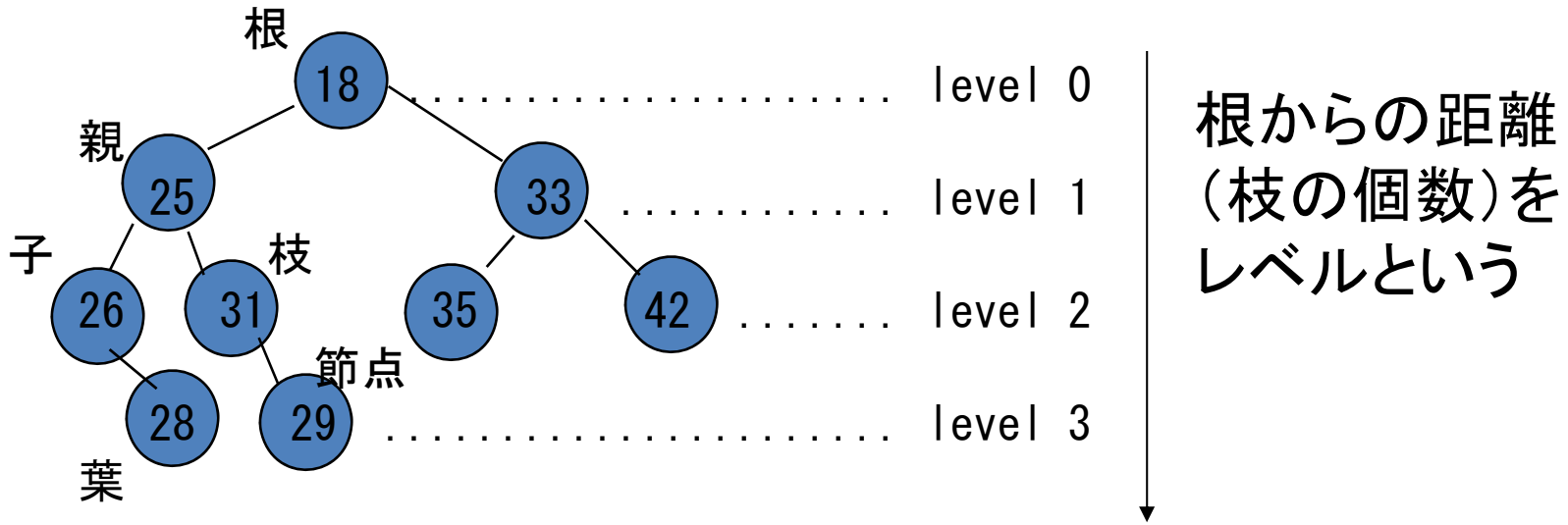
- 格納: O(1)

```
heap[top++]=x
```

- 取り出し: O(n)

```
m = 0;
for(i=1; i<top; i++)
  if(heap[i] < heap[m])
    m = i;
x = heap[m];
heap[m] = heap[top-1];
top = top - 1;
return x;
```

2分木によるheapの実現

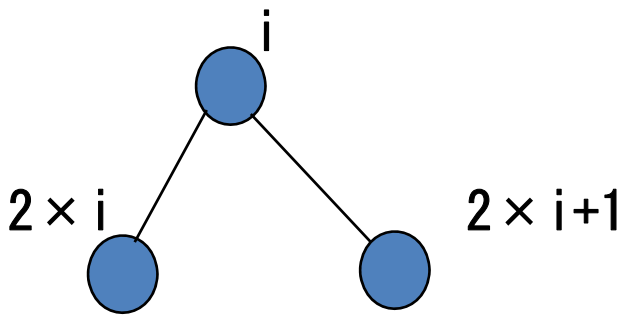


根:親のない節点
葉:子のない節点

どの節点も2個以内の子をもつとき2分木という

Heapを実現する2分木の性質

1. 根に番号1を割り当てる
2. 番号 i の節点の左の子には 2 × i、右の子には 2 × i+1 の番号を割り当てる.

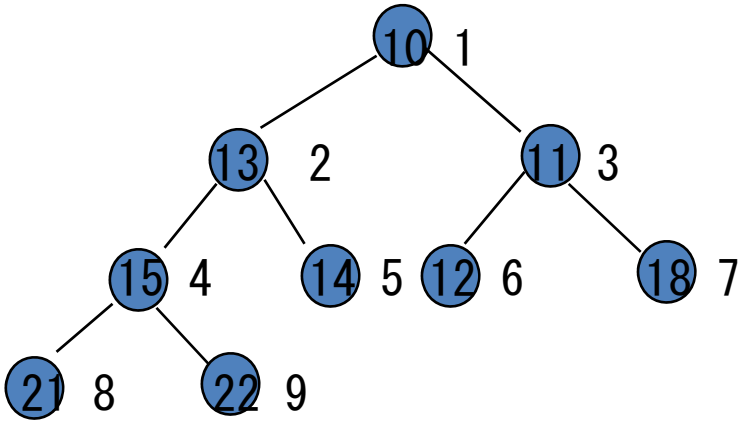


3. 節点の個数nを超える番号をもつ節点は存在しない.
4. どの枝についても，親には子以下のデータを蓄える

ヒープの最大レベル: $\mathrm{ceil}(\log_2(n+1)-1)$

どの節点についても根から唯一のパスが存在して
その長さはO(log n)

Heapを実現する2分木の性質: 例



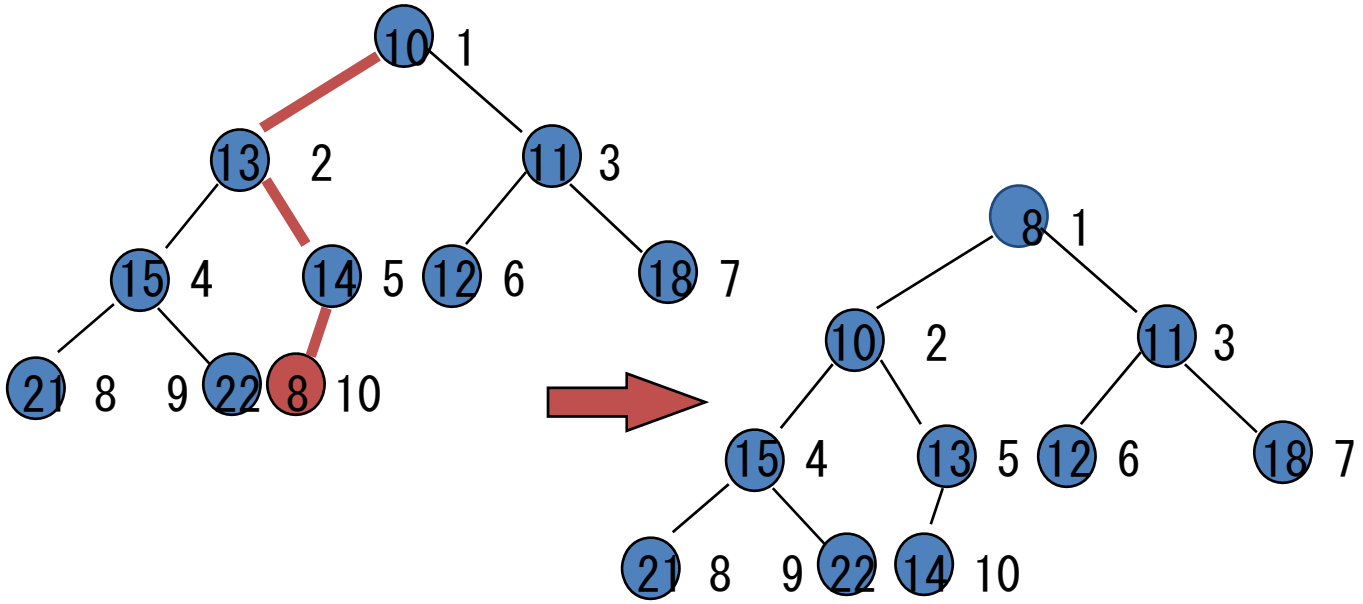
1. 根に番号1を割り当てる
2. 番号 i の節点の左の子には 2×i、右の子には 2×i+1 の番号を割り当てる.
3. 節点の個数nを超える番号をもつ節点は存在しない.
4. どの枝についても，親には子以下のデータを蓄える

連結リストを使うことなく配列で表現可能:

1	2	3	4	5	6	7	8	9
10	13	11	15	14	12	18	21	22

Heapへのデータの追加

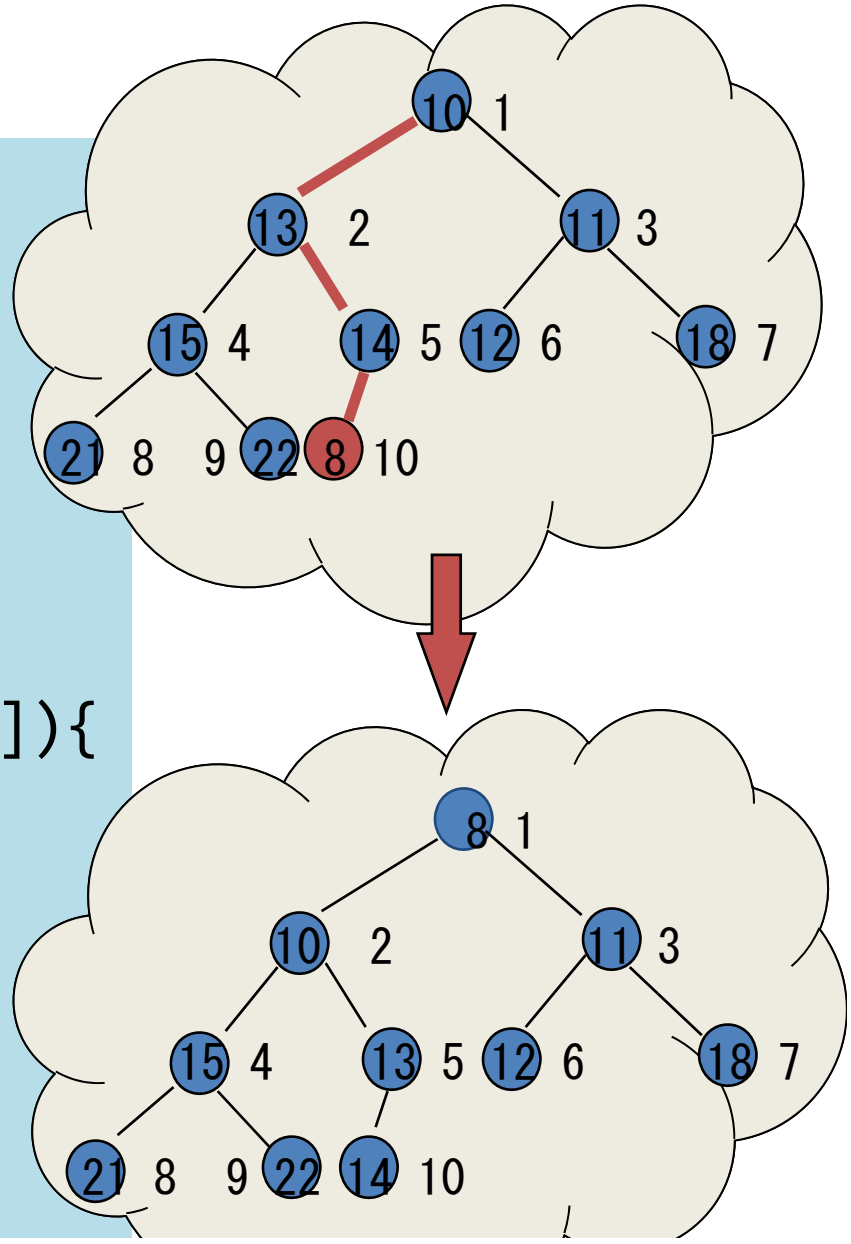
(1)データを節点 n+1 に格納（配列の n+1 番目）
(2)根に向けて上へ上へとたどり、
　if 親のデータ ＞ 子のデータ then 親子のデータを交換



n+1番目の節点から根までの経路上の要素を大きさの順に並べる。このとき残りの部分との間で矛盾を引き起こすことはない。

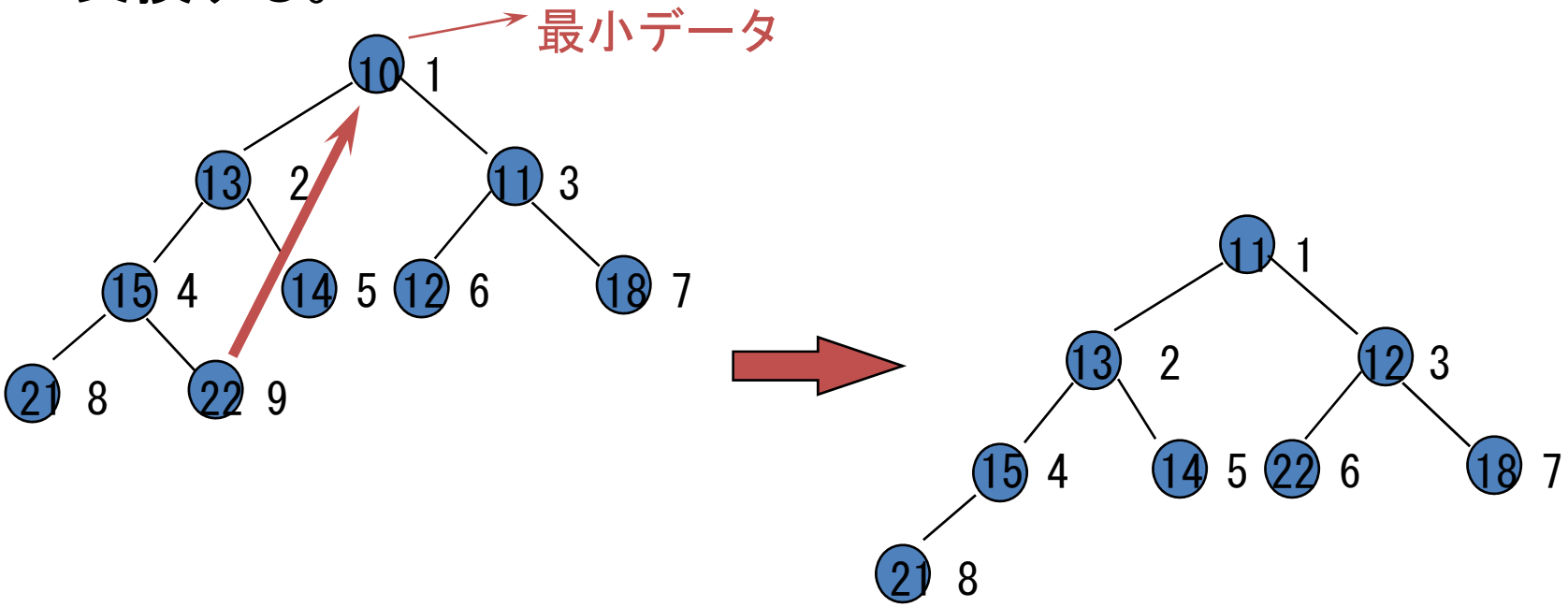
Heapへデータを追加する プログラム

```
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=i/2;
    }
    heap[i] = x;
  }
}
```



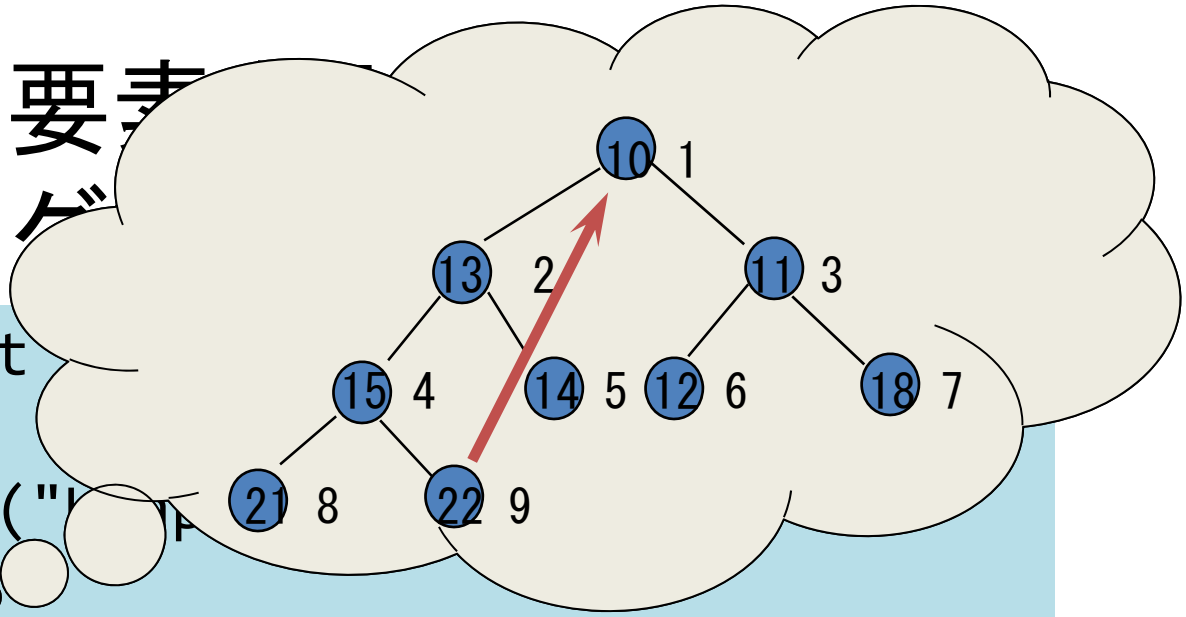
Heap: 最小要素の取り出し

(1)根のデータを最小データとして取り出す
(2)番号nの節点のデータを根に移動
(3)根から葉に向けてたどる
　このとき、2個の子節点のうちデータの小さい方を選び、それが親のデータより小さいときはデータを交換する。

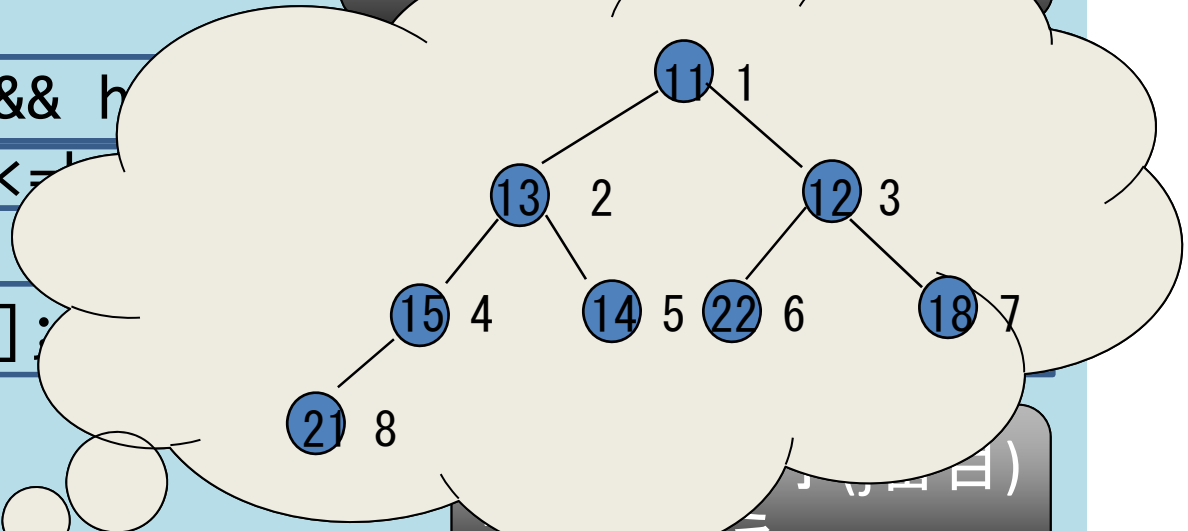


Heapの最小要素
プログ

```
int* deleteMin(int
  int x, i, j, t;
  if(n == 0) stop("
  else{
    heap[1]=heap[n--];
    for(i=1;i*2<=n;i=j){
      j=i*2;
      if(j+1<=n &&
      if(heap[i]<=
      else {
        t=heap[i];
      }
    }}
  return heap;}
```



2分木のiに子がある&&
右に最小値の可能性あり



を入れ替え

2分heapの計算時間

- heapのサイズをnとする
  - 追加: O(log n)
  - 取り出し: O(log n)

    - どちらの操作も明らかにヒープの深さに比例する時間で実行
    - ヒープの深さはO(log n)