データ構造と アルゴリズムＩＢ

第4回

データ構造

- データをどのような形式で格納し，どのように利用するかを与える
- 現実のプログラミングでは，アルゴリズムの選択よりもデータ構造の選択が重要であることが多い
  ⇒ データ構造が決まれば用いるべきアルゴリズムも決まる

367

データ構造の意味

- 今回紹介するデータ構造は，配列や構造体等の，一般的で柔軟なデータ構造を用いて定義している
- では，配列や構造体があれば十分? なぜわざわざ制限されたデータ構造を考える必要がある?
- データ構造へのアクセス／変更の方法を，あえて制限することに意味がある！
- 自分で定義したデータ構造を用いる／薄皮を被せることにより，データ構造への予期せぬアクセス／変更が生じることを防ぎ，プログラムの可読性／安全性／保守性が高まる

368

集合を扱うデータ構造

- 集合：数学，計算機科学において基本的
- 動的集合：要素が追加/削除/変更される
- 集合に対して行う操作によってデータ構造を変える
- 行いたい操作によって最適なデータ構造が決まる

行う操作	データ構造
挿入，削除， 存在判定	辞書
挿入 最小要素の取出し	プライオリティー キュー

369

動的集合の基本

- 各要素はオブジェクトとして表現される
- オブジェクトはキーと付属データからなる
- 集合の操作で扱うフィールドがあってもよい
  – 他のオブジェクトへのポインタなど
- キーは全順序を持つとする場合もある

370

動的集合に関する操作

1. 集合に関する情報を返す質問 (query)
   - SEARCH(k): $key[x] = k$ である S の要素 x へのポインタを返す．存在しなければ NIL.
   - MINIMUM(): 全順序集合 T において，最小のキーを持つ要素を返す
   - MAXIMUM(): 全順序集合 T において，最大のキーを持つ要素を返す
   - SUCCESSOR(x):全順序集合 T においてキーが x のキーの次に大きな要素を返す．x が最大なら NIL.
   - PREDECESSOR(x): キーが x のキーの次に小さな要素を返す．x が最小なら NIL.

371

動的集合に関する操作

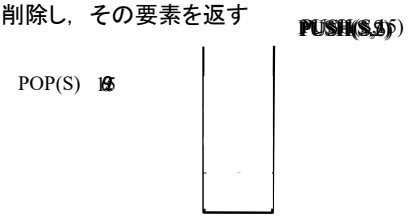
2. 集合を変える修正操作 (modifying operation)
   - INSERT(x): 集合 S に要素 x を加える.
   - DELETE(x): x へのポインタが与えられたとき，S から x を取り除く.
- SUCCESSOR, PREDECESSORは同じキーが複数ある集合にも拡張される
- 集合操作を実行するのにかかる時間は集合のサイズで測る

10.1 スタックとキュー

- 動的集合，挿入と削除をサポートする
- スタック (stack) では，DELETEでは最後に挿入された要素が取り除かれる
  - 後入れ先出し (last-in, first-out; LIFO) という
- キュー (queue) では，最初に挿入された要素が取り除かれる
  - 先入れ先出し (first-in, first-out; FIFO) という

スタック (Stack)

- INSERT, DELETEの代わりにPUSH, POPと呼ぶ
  - PUSH(S, x): スタックSに要素 x を加える.
  - POP(S): スタックから最後に PUSH された要素を削除し，その要素を返す



配列によるスタックの実装

- 最大 MAX 要素を格納できるスタックを配列S[1..n]で実装
- 最後に挿入された要素の位置を記録する属性をS.topとする
- 要素は S[1..S.top] に格納される
  - S[1]: スタックの底
  - S[S.top]: スタックの最上部

実装例

- PUSH(S,x)
  - topを1増やし，x を配列に入れる
  - O(1) 時間

```
PUSH(S, x)
 S.top = S.top + 1
 S[S.top] = x
```

- POP(S)
  - スタックが空ならエラー
  - サイズを1減らし，最上部の要素を返す
  - O(1) 時間

```
POP(S)
 if (STACK_EMPTY(S))
  error "アンダーフロー"
 else S.top = S.top - 1
      return S[S.top + 1]
```


その他の関数

- STACK_EMPTY(S)
  - スタックが空なら1を返す
  - O(1) 時間

```
STACK_EMPTY(S)
 if S.top == 0 return TRUE
 else return FALSE
```

課題

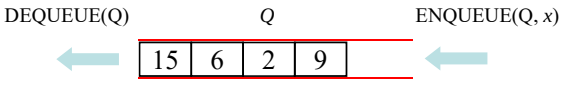
- n要素の配列A[1,…,n]を用いて，二本のスタックを実装したい
- 二つのスタックの要素の合計がnを超えない限り，どちらもオーバフローしないようにするにはどうしたらよいか?

答え

- 一つのスタックは最初から詰めて，もう一つのスタックは後ろから逆に詰める
- topが同じになったらオーバーフロー

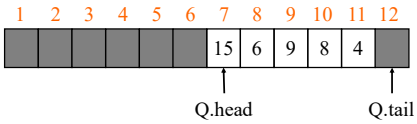
キュー (Queue)

- INSERT, DELETEの代わりにENQUEUE, DEQUEUEと呼ぶ
- 人の並んだ列と同じ
  - ENQUEUEでは列の最後に追加
  - DEQUEUEでは列の先頭を取り出す



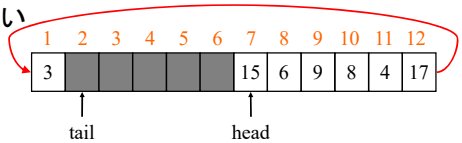
配列によるキューの実装

- 最大 MAX–1 要素を格納できるキュー
  - Q: 要素を格納する配列，以下の属性を持つ
  - Q.length: 配列の長さ
  - Q.head: キューの先頭の位置
  - Q.tail: 次に追加される位置



環状バッファ (Ring Buffer)

- 配列 Q の右端と左端はつながって輪になっていると考える
- Q[Q.length] の右は Q[1] だとみなす
- 要素は Q[Q.head..Q.length] , Q[1..Q.tail–1] に格納される
- DEQUEUEの際に全要素を左にずらす必要がない



実装例

- ENQUEUE(Q, x)
  - x をQ[tail]に入れる
  - tailを1増やす
  - O(1) 時間

```
ENQUEUE(Q, x)
 Q[Q.tail] = x
 if Q.tail == Q.length
      Q.tail = 1
 else Q.tail = Q.tail + 1
```

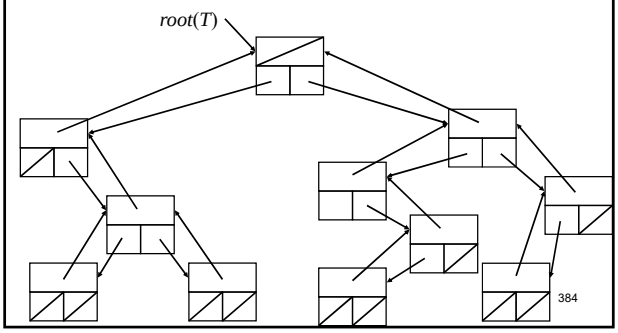
- DEQUEUE(Q)
  - Q[Q.head]を取り出す
  - Q.headを1増やす
  - O(1) 時間

```
DEQUEUE(Q)
 x = Q[Q.head]
 if Q.head == Q.length
      Q.head = 1
 else Q.head = Q.head + 1
 return x
```

注: オーバーフロー, アンダーフロー処理は省略してある

10.4 根付き木の表現

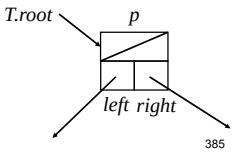
- （二分）根付き木をオブジェクトを用いて表現する



384

2分木

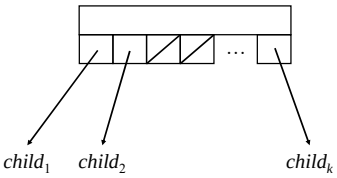
- 各節点xは属性 *p*, *left*, *right*, *key*を持つ
  - *x.p*: 親へのポインタ，NILなら根
  - *x.left*: 左の子へのポインタ，NILなら子を持たない
  - *x.right*: 右の子へのポインタ，NILなら子を持たない
- *T.root*: *T* の根を指す．NILなら木は空



385

*k*分木

- 子供を最大 *k* 個持つ木の表現
  - *left*, *right*の代わりに $child_1$, $child_2$,..., $child_k$ とする
  - 子供の数が一定でないと記憶領域を無駄にする

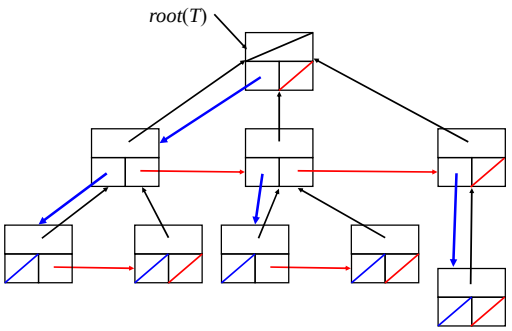


386

左-子, 右-兄弟表現
(Left-Child Right-Sibling Representation)

- *k* 分木を2分木で表現する方法
- 任意の *n* 節点の根付き木を O(*n*) 領域で表現可能
  - *x.p*: 親へのポインタ
  - *x.left-child*: *x* の子で最も左にあるものへのポインタ
  - *x.right-sibling*: *x* の右の兄弟へのポインタ
- *x* が子を持たないとき x.*left-child* = NIL
- *x* がその親の右端の子のときx.*right-sibling* = NIL

387



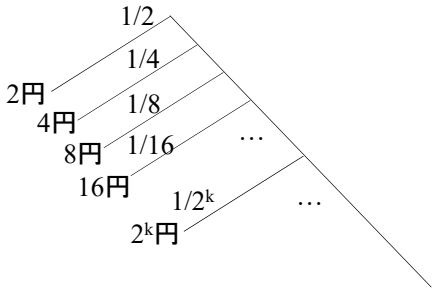
388

セント・ペテルスブルグの逆説

- 以下のようなクジを考える．
  - コインを投げる．
  - 表が出たら終わり．2円もらえる．
  - 裏が出たらならもう一回コインを投げる．
  - 表が出たら終わり，4円もらえる．
  - 裏が出たならもう一回コインを投げる．
  - 表が出たら終わり，8円もらえる．
  - 以下繰り返し，k回目に初めて表が出たら 2^k円もらえる
- このクジに参加するのに，いくらまでなら払っても良いか?

389

二分木で表すと?



セント・ペテルスブルグのパラドックス

- なぜ期待値が無限大のクジに，10円払うのもいやなのか?
  - リスクに対する態度：人間はリスクを避けたいと思う --- 期待値が小さくても，確実な方を好む
    - 一億円確実にもらえるのと，コインを投げて表なら二億円，裏なら0円のクジの価値は同じ?
  - 金額が倍になっても，うれしさは倍にならない：二万円は一万円の倍うれしいが，二兆円は一兆円の倍ほどはうれしくない
  - このクジは現実には成立し得ない（地球全体の富の総額を超える賞金を約束している）

リスクに対する態度

- 自動車保険
  - 保険に入らない場合：ほとんどの場合にコスト0，非常に低い確率pで，事故を起こして1億円払うというクジを引く
  - 保険に入る場合：確実に5万円損をする代わりに，上のクジを引かなくてよい
  - p × 一億円 < 5万円
  - 期待値だけを考えるなら，保険に入らない方が良い
  - 通常，人はリスク回避的，しかし，保険に入る人が宝くじを買うこともある (100円払って，期待利得が50円未満のものを買う)

10. 2 連結リスト

リストの利点：

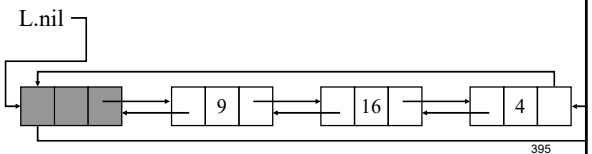
- 何個の要素が入るか あらかじめ予想できない場合でも使える
- 最悪の場合を予想して，大きめに領域を取っておく必要がない
- 再帰的な構造をしているので，再帰的なプログラムと相性が良い

リストの種類

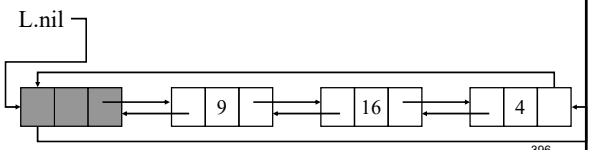
- 一方向 (singly linked) と双方向 (doubly linked)
  - 一方向のとき，各要素は prev を持たない
- 既ソート (sorted) と未ソート
  - 既ソート: リストの線形順序はキーの線形順序に対応
  - 未ソート: 任意の順序
- 循環 (circular list) と非循環
  - 循環: リストの先頭要素の prev はリストの末尾を指し，末尾の next はリストの先頭を指す
- 以下では未ソート双方向循環リストを扱う

連結リスト (Linked Lists)

- オブジェクトをある線形順序に並べて格納するデータ構造
- 連結リストでの線形順序は，オブジェクトが含むポインタで決定される
- 双方向連結リスト (doubly linked list) *L* の要素
  - キーフィールド KEY
  - ポインタフィールド PREV, NEXT
- リストLの最初の要素はキーを持たない特別なオブジェクト（L.nil）



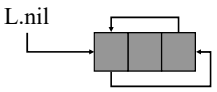
- 双方向リストはL.nilを含めた循環リストで表現される
- 先頭要素は L.nil.next
- x.next: リスト中の *x* の直後の要素のポインタ
- x.next = L.nil のとき，*x* は最後の要素
- x.prev: *x* の直前の要素のポインタ
  - x.prev = L.nilのとき，*x* はリストの最初の要素



396

空リスト

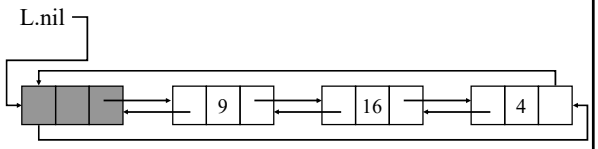
- 以下で表現される



397

連結リストからの削除

```
LIST_DELETE(L, x)
  x.prev.next = x.next
  x.next.prev = x.prev
```

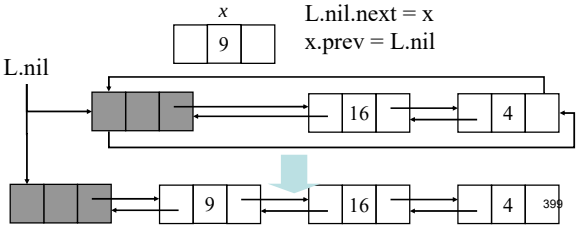


398

連結リストへの挿入

- LIST_INSERT(L,*x*): *x* を リストの先頭に挿入
  - *x* のキーは既にセットされているとする
- O(1) 時間

```
LIST_INSERT(L, x)
  x.next = L.nil.next
  L.nil.next.prev = x
  L.nil.next = x
  x.prev = L.nil
```

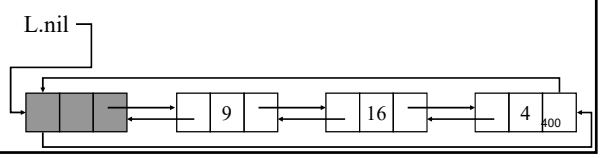


399

連結リストの探索 (再帰バージョン)

- R_LIST_SEARCH(*L, k, i*): リストLに属する，要素 *i*以降でキー *k* を持つ最初の要素のポインタを返す
- R_LIST_SEARCH(*L, k,* L.nil.next) とすれば最初から探索

```
R_LIST_SEARCH(L, k, i)
 if i ≠ L.nil and i.key≠ k
   return R_LIST_SEARCH(L, k, i.next)
 else return i
```

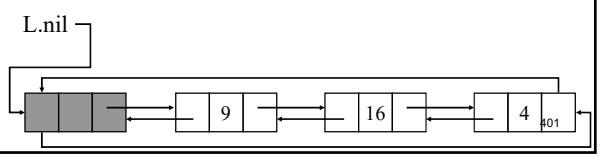


400

最小値の探索 (再帰バージョン)

- R_MINIMUM(L, i): リストに属する，要素 i 以降の最小値を返す
- R_MINIMUM(L, L.nil.next) とすれば全体の最小値を返す

```
R_MINIMUM(L, i)
 if i.NEXT(i) ≠ L.nil
   return i.KEYとR_MINIMUM(L, i.next)の最小値
 else return i.key
```



401

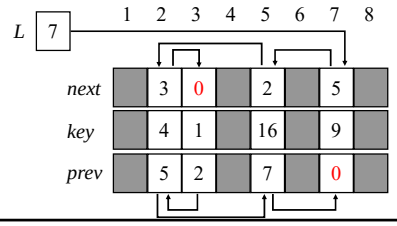
10.3 ポインタとオブジェクトの実現

- オブジェクトは構造体やポインタを用いて表現される
- これらを標準で持たない言語 (FORTRAN等) で実現するにはどうすればいいか?
- いつでもCやJAVAが使えるとは限らない
  - 組込みソフトウェアの開発等

402

オブジェクトの多重配列表現

- オブジェクトのフィールドごとに配列を用いる
- ポインタの代わりに，配列の添字を用いる
- 0 は NILを表す
- すべてのオブジェクトを連結（非循環）リストで管理
- 変数 *L* はリストの先頭を表す



403

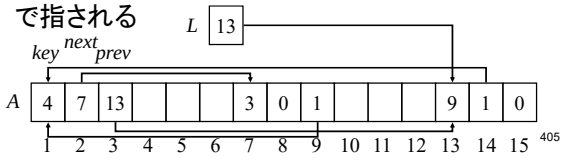
多重配列表現の欠点

- 多くの操作ではオブジェクト中の複数のフィールドを読み書きする
  - *key*, *next*, *prev*
- 多重配列表現では，CPUキャッシュが利きにくい
  - 1つのオブジェクトがメモリ中で分断されているため

404

オブジェクトの単一配列表現

- 計算機のメモリは大きな配列とみなせる
  - 0 番地から *M*–1 番地
- オブジェクトはメモリのある連続した場所を占める
- ポインタはオブジェクトの占める場所の先頭番地を指す
- 各フィールドは，ポインタにオフセットを加えたもので指される



405