基本的なデータ構造

計算科学研究センター
システム情報工学研究科CS専攻
北川博之

配列

- 最も基本的な物理構造
- データを格納するための領域が連続的に配置されたもの
- インデックス（添字）：配列の何番目かを示す番号
- 要素：配列に格納さえる各データ
- 通常は，一つの配列の要素は全て同一の型．

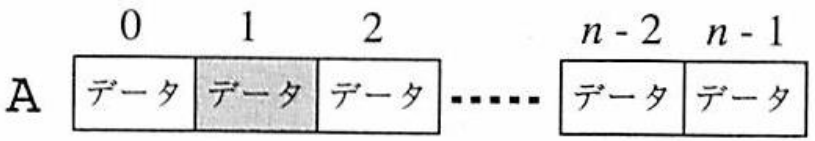


図 2.1　1 次元配列

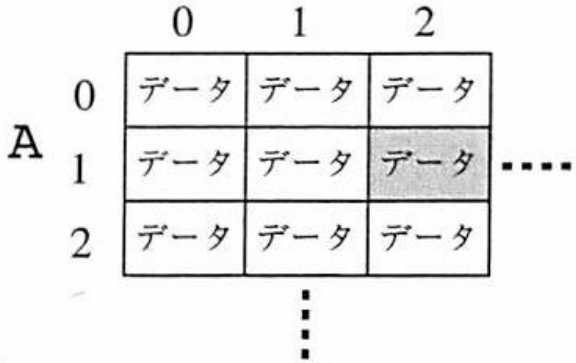
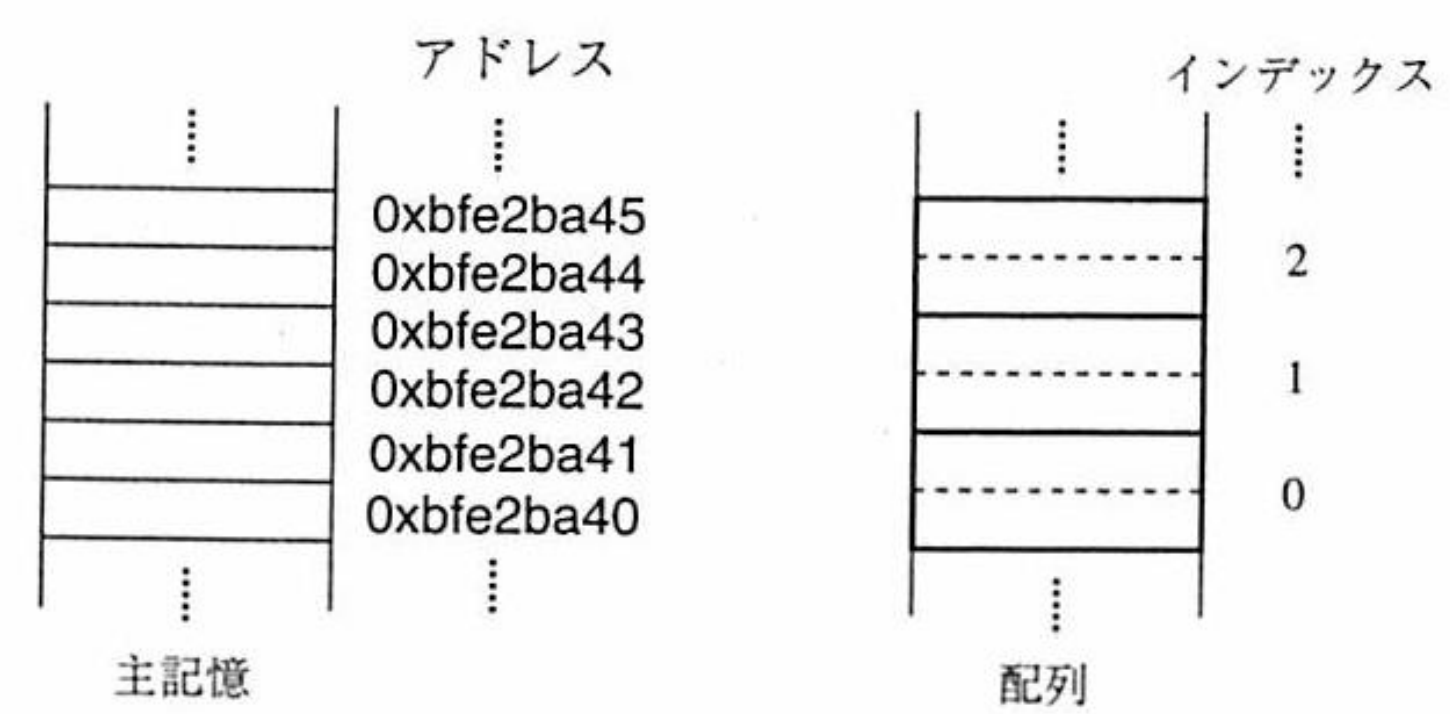


図 2.2　2 次元配列

配列

- 配列はコンピュータの主記憶構造に近いデータ構造



配列に対する操作

- あるインデックスの要素の参照
  - 要素が格納されているアドレスを計算して該当要素を参照：O(1).
- あるインデックスの要素への格納，書換え
  - 上記と同様にO(1).
  - ただし，途中にデータを割り込ませる場合は，その位置以降のデータを1個ずつ後ろにずらすため平均O(n).

リスト 2.1 配列へのデータの挿入

```
// n:格納されている要素数
insert_array(A, k, x) {
  // A:配列名、k:挿入位置のインデックス、x:挿入するデータ
  for (i = n; i > k; i = i-1)
    A[i] = A[i-1];
  A[k] = x;
}
```

リンク配置

- 配列と並ぶ基本的な物理構造
- セル(cell)と呼ぶデータの記憶単位が主記憶上に分散されて配置される.
- データ部:データ自身を格納
- ポインタ部:次のセルをつなぐためのデータ. 次に続くセルが配置されている主記憶上のアドレスを格納.

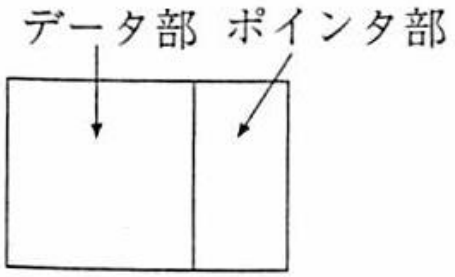


図 2.3 セルの構造

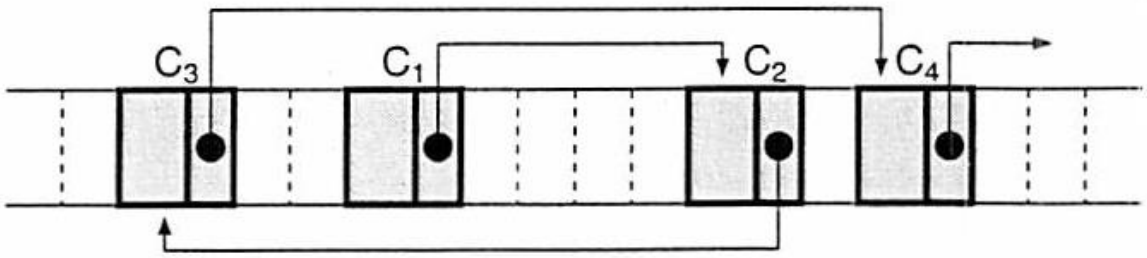
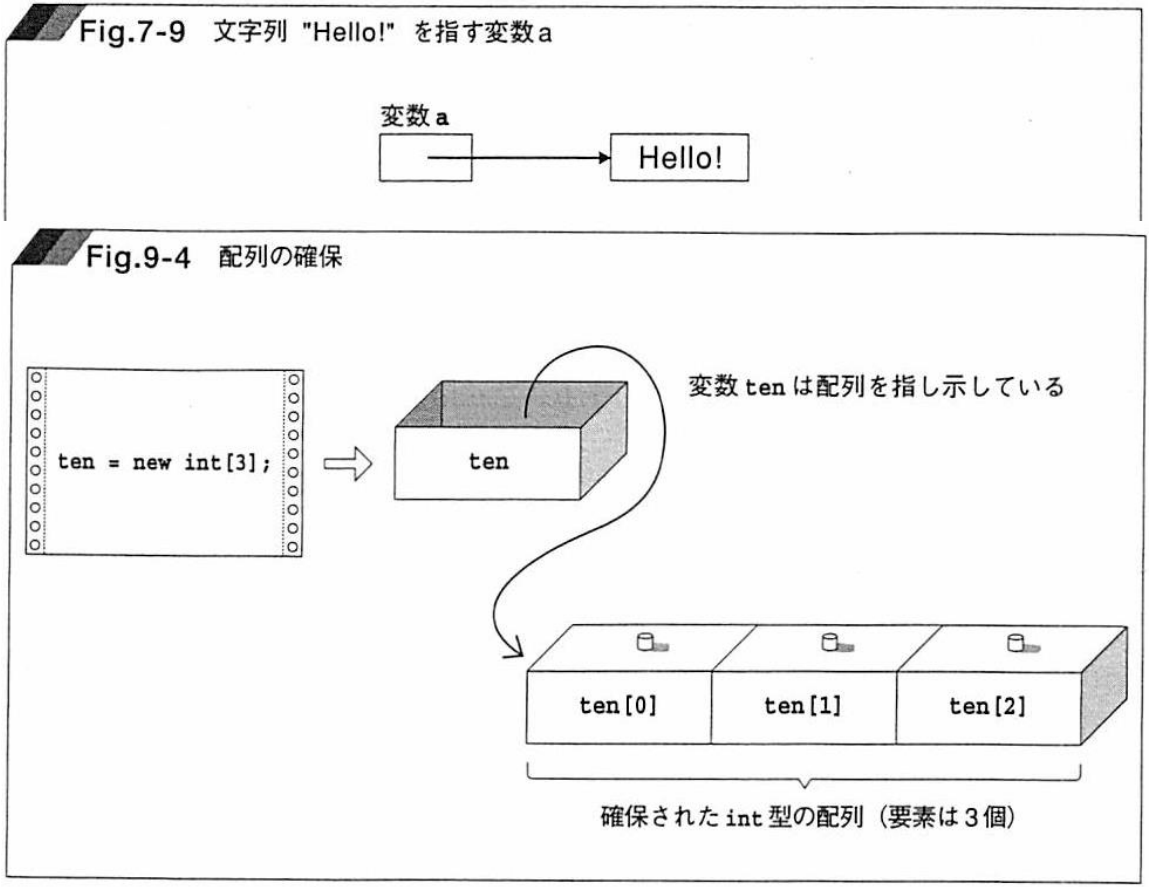


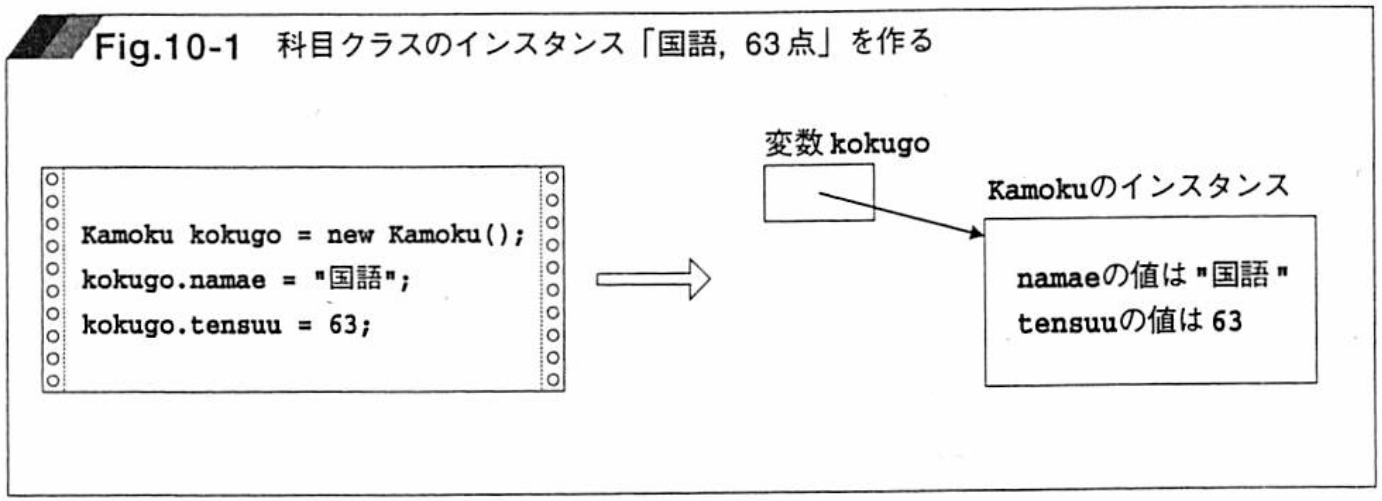
図 2.4 主記憶上のセルの配置の様子

ポインタ

- Javaではポインタを直接は扱わないが，参照型（文字列，配列，クラス等）の変数は，内部的には，その値を格納した場所へのポインタを保持している．



ポインタ



- あるクラスのフィールドがクラスを参照すれば，内部的にはポインタの連鎖ができる．

連結リスト

- セルが1次元的に数珠つなぎになったもの
- 線形リスト，リストともいう

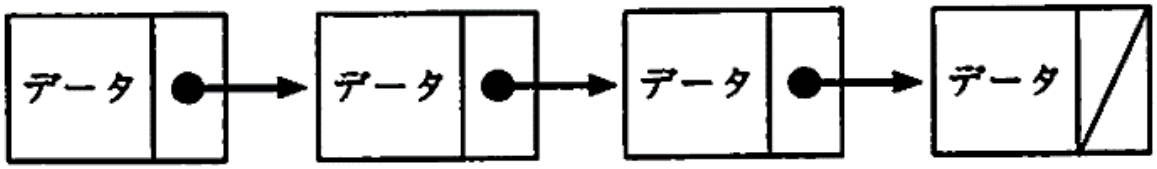


図 2.5　連結リスト

- 先頭セルのアドレスの保持の仕方

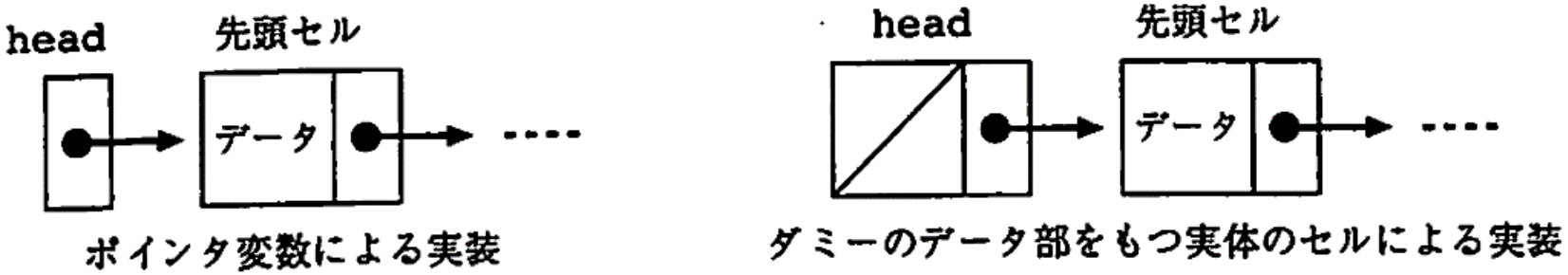
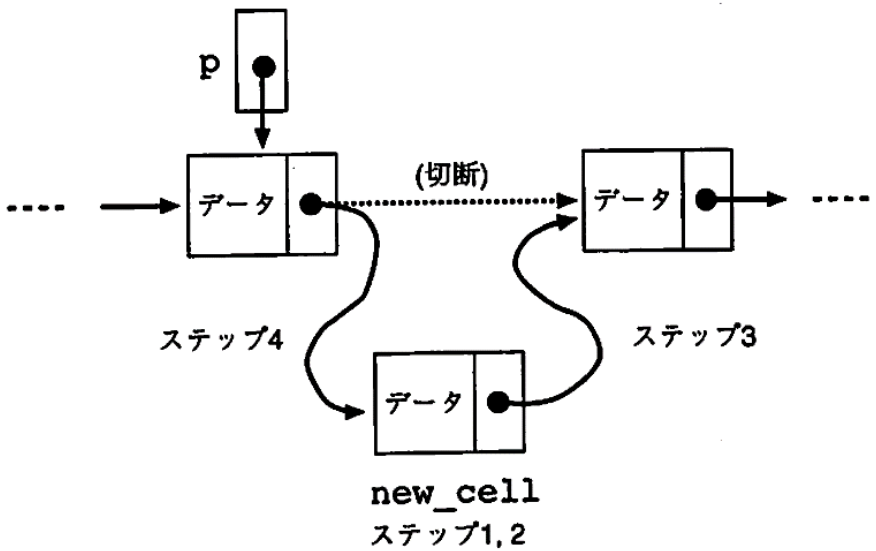


図 2.6　先頭セルのアドレスを保持する head の 2 つの実装方法

連結リストの基本操作

- 新しいセルの挿入
  - 時間計算量：O(1)



リスト 2.2 新しいセルの挿入

```
// 新しいセルの挿入
//      挿入位置はセルへのポインタ p が指すセルの直後
  insert_cell(p, d) {// p:セルへのポインタ
                     // d:挿入するセルに格納するデータ
1:  新しいセル new_cell の作成;
2:  new_cell のデータ部 =  d;
3:  new_cell のポインタ部 = p が指すセルのポインタ部;
4:  p が指すセルのポインタ部 = new_cell のアドレス;
  }
```

連結リストの基本操作

- 先頭への新しいセルの挿入
  （headをポインタ変数で実装した場合）
  - headをダミーのデータ部をもつ実体のセルとした場合は，
    リスト2.2のままで問題ない．

リスト 2.3 連結リストの先頭への新しいセルの挿入（headをポインタ変数で実装した場合）

```
// 新しいセルの挿入（挿入位置は連結リストの先頭）
// head:先頭セルへのポインタ
insert_cell_top(d) {// d:挿入するセルに格納するデータ
  新しいセル new_cell の作成;
  new_cell のデータ部 = d;
  new_cell のポインタ部 = head;
  head = new_cell のアドレス;
}
```

連結リストの基本操作

- セルの削除
  - 時間計算量：O(1)

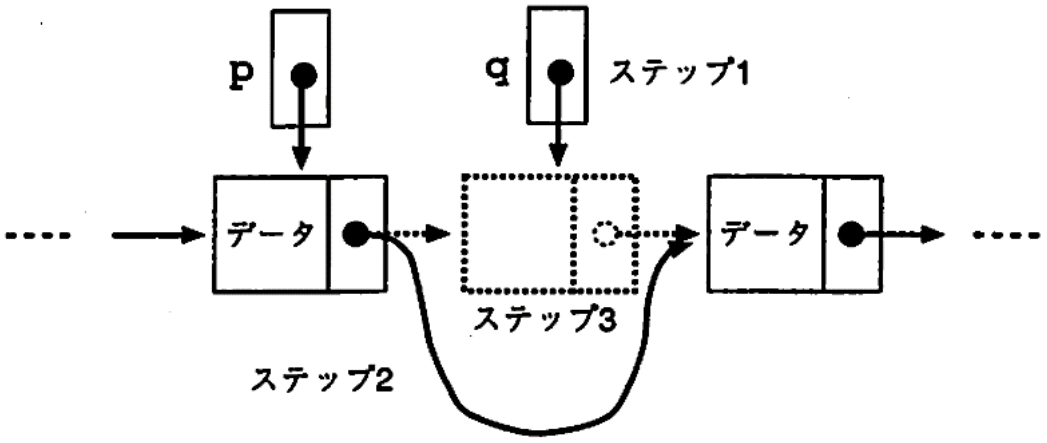


図 2.8　連結リストからセルを削除する手順

リスト 2.4 セルの削除

```
// セルの削除
// 削除されるのはセルへのポインタ p が指すセルの直後のセル
  delete_cell(p) {// p:セルへのポインタ
1:  セルへのポインタ q = p が指すセルのポインタ部;
2:  p が指すセルのポインタ部 = q が指すセルのポインタ部;
3:  q が指すセルが使用している主記憶領域を開放;
  }
```

連結リストの基本操作

- 先頭セルの削除
  （headをポインタ変数で実装した場合）
  - headをダミーのデータ部をもつ実体のセルとした場合は，リスト2.4のままで問題ない．

リスト 2.5 連結リストの先頭セルの削除（head をポインタ変数で実装した場合）

```
// 連結リストの先頭セルの削除
// head:先頭セルへのポインタ
delete_cell_top() {
  セルへのポインタ q = head;
  head = q が指すセルのポインタ部;
  q が指すセルが使用している主記憶領域を開放;
}
```

連結リストの基本操作

- セルの探索
  - 指定した値をデータ部にもつセルを探索
  - 連結リストの先頭から順番にデータ部を調べる
  - 時間計算量：O(n)

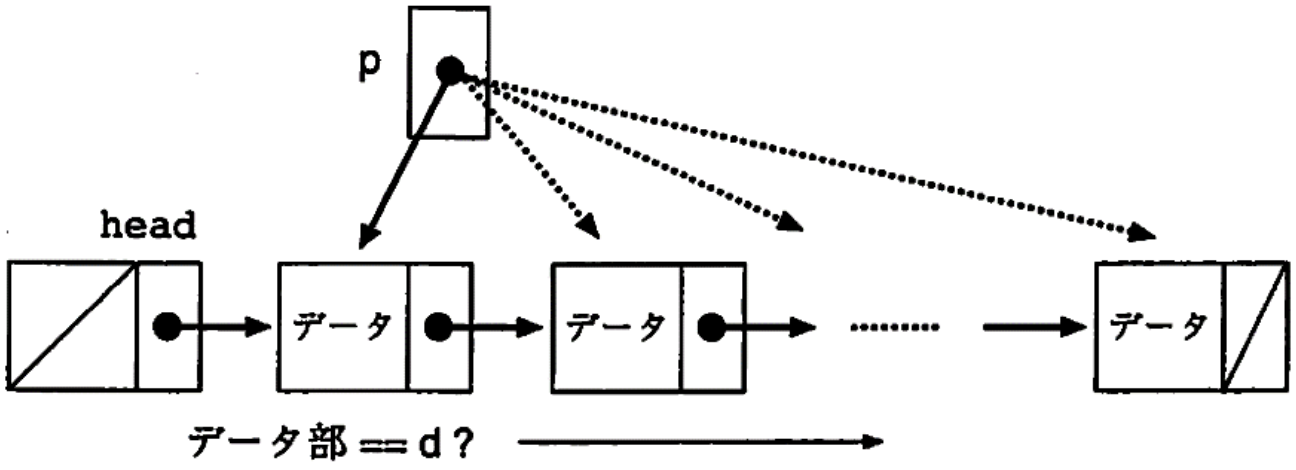


図 2.9　セルの探索

連結リストの基本操作

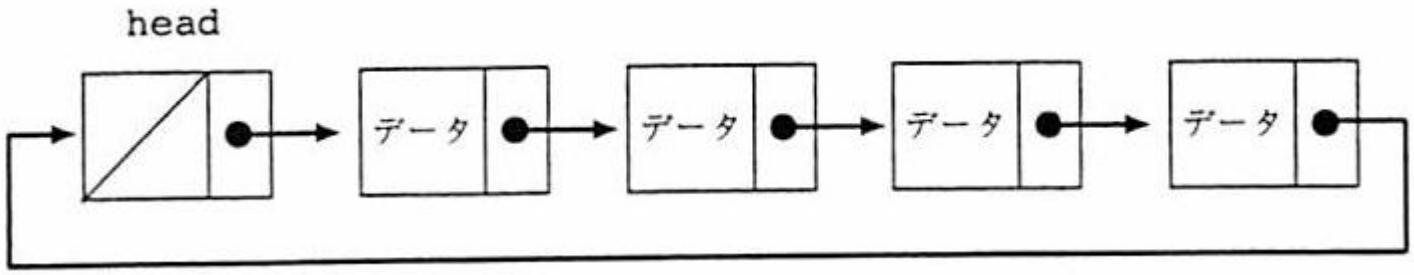
- セルの探索

リスト 2.6 セルの探索

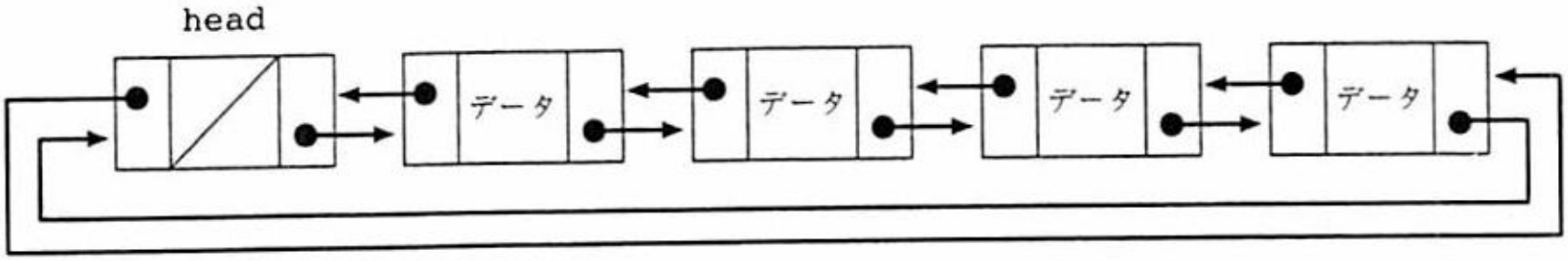
```
// セルの探索
//    セルへのポインタ p が指すセルを始点としてデータ部が d
//    に一致するセルを探索し，見付かった場合はそのアドレス
//    を，見付からなかった場合は NULL を返す
search_cell(p, d) {// p:セルへのポインタ
                   // d:探索するデータ
  while (p != NULL) {
    if (p が指すセルのデータ部 == d) return p;
    p = p が指すセルのポインタ部;
  }
  return NULL;
}
```

連結リストの変形

- 循環リスト(circular list):連結リストの末尾のセルのポインタがリストの先頭のセルを指すようにしたもの



- 双方向リスト(doubly-liked list):直後だけでなく直前のセルへのポインタを加えたもの



スタック(stack)

- 上部が空いた容器のようなもの
- LIFO(Last In First Out):後に格納されたデータほど先に取り出される
- 基本操作
  - プッシュ(push)
  - ポップ(pop)

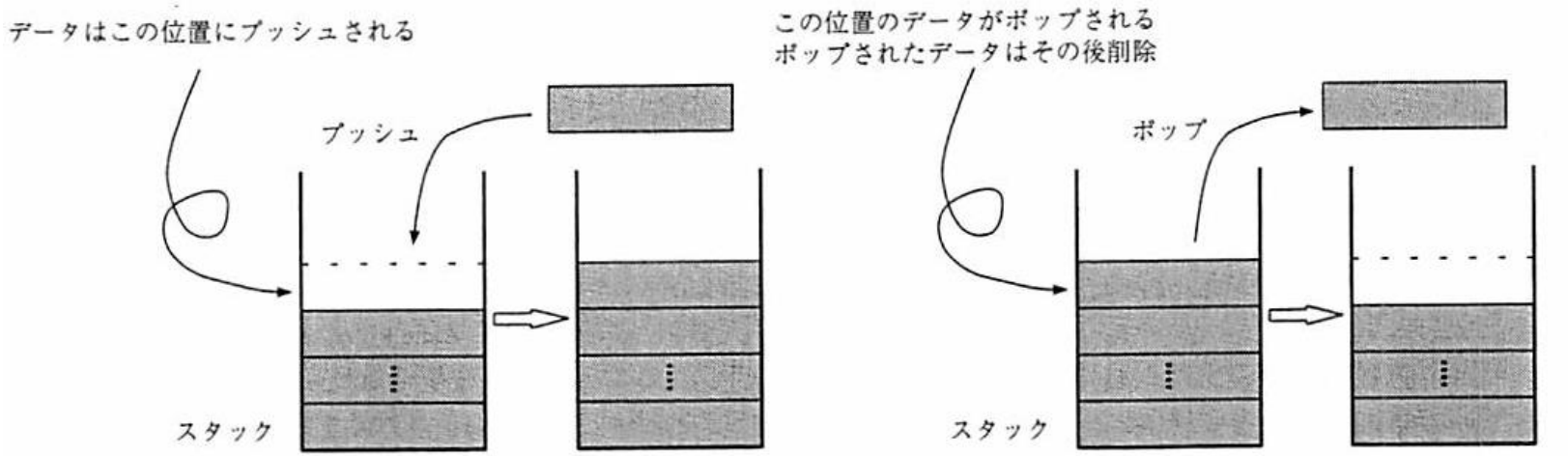


図 2.10　スタックの基本操作

スタックの実装

- １次元配列を用いた実装
  - STACK：配列名
  - n：配列のサイズ
  - top：スタックポインタ，データが格納されている１番上の要素のさらにその一つ上の要素のインデックス

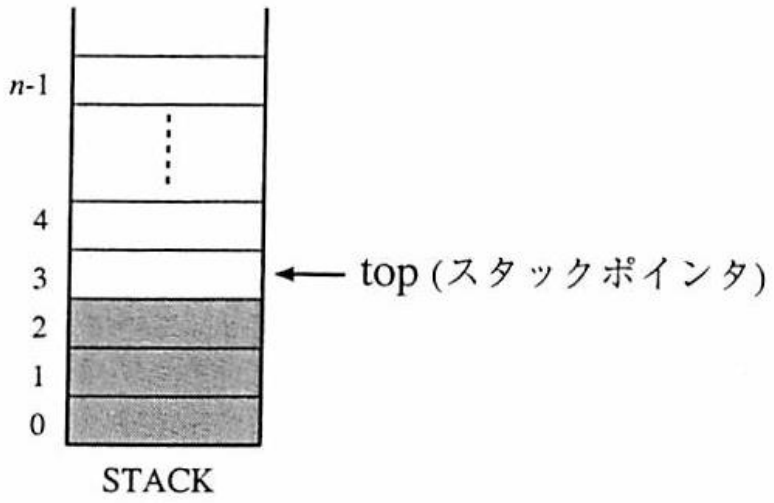


図 2.11　１次元配列によるスタックの実装

スタックの実装

- プッシュ
  - プッシュするデータをtopの位置に格納
  - オーバーフロー：スタックが満杯（top=n)の場合
  - 時間計算量：O(1)

リスト 2.7 スタック操作 — プッシュ

```
// スタックへのデータのプッシュ
// top:スタックポインタ, n:スタック用の配列のサイズ
push(STACK,d) {// STACK:スタック（配列名）
                // d:プッシュするデータ
  if (top > n-1)
    "Stack Overflow !!" と出力して終了;
  STACK[top] = d;
  top = top+1;
}
```

スタックの実装

- ポップ
  - topを1減らしてその位置のデータを出力
  - アンダーフロー：スタックが空（top=0)の場合
  - 時間計算量：O(1)

リスト 2.8 スタック操作 — ポップ

```
// スタックからのデータのポップ
// top:スタックポインタ
pop(STACK) {// STACK:スタック（配列名）
  if (top == 0)
    "Stack Underflow !!" と出力して終了;
  top = top-1;
  return STACK[top];
}
```

キュー(queue)

- 待ち行列
- FIFO(First In First Out):先に格納されたデータほど先に取り出される
- 基本操作
  - エンキュー(enqueue)
  - デキュー(dequeue)

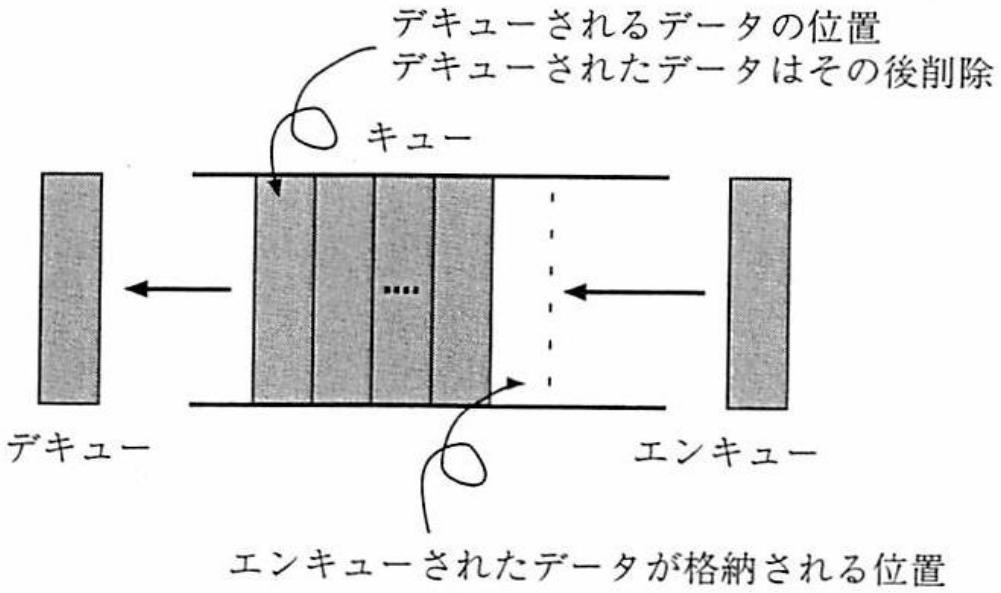


図 2.12　キューの基本操作

キューの実装

- １次元配列を用いた実装
  - QUEUE：配列名
  - n：配列のサイズ
  - front：デキューされるデータの位置を示すインデックス
  - rear：エンキューされるデータを格納する位置を示すインデックス

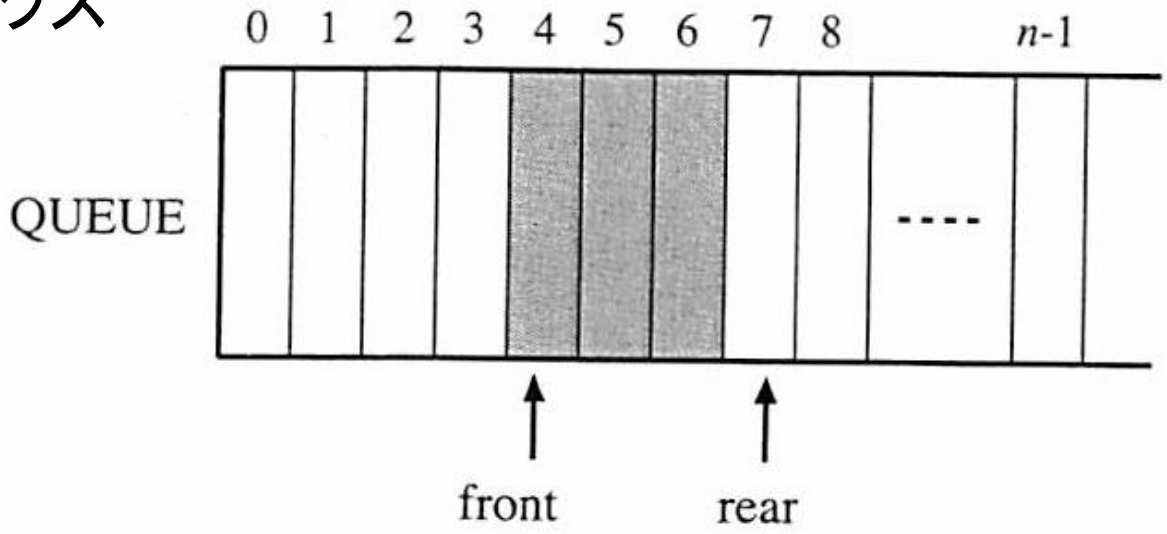


図 2.13　配列によるキューの実装

キューの実装

- エンキュー
  - データをrearの位置に格納
  - オーバーフロー：rear=nの場合
  - 時間計算量：O(1)

リスト 2.9 キュー操作 — エンキュー

```
// キューへのデータのエンキュー
// rear:データの格納位置を示すポインタ
// n:キュー用の配列のサイズ
enqueue(QUEUE,d) {// QUEUE:キュー（配列名）
                  // d:エンキューするデータ
  if (rear > n-1)
    "Queue Overflow !!" と出力して終了;
  QUEUE[rear] = d;
  rear = rear+1;
}
```

キューの実装

- デキュー
  - frontの位置のデータを出力
  - アンダーフロー：キューが空(front=rear)の場合
  - 時間計算量：O(1)

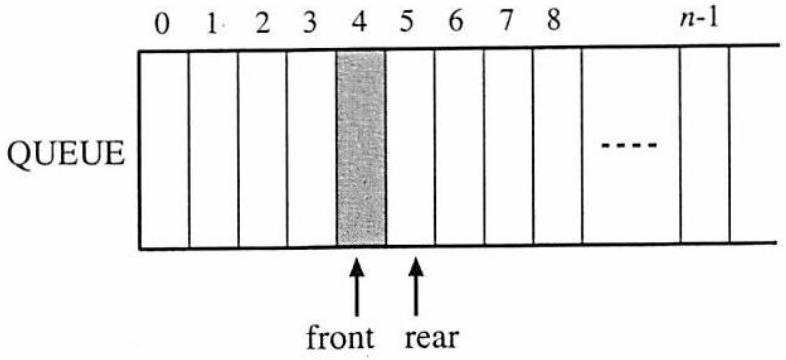


図 2.14 データが 1 つだけ格納された状態のキュー

リスト 2.10 キュー操作 — デキュー

```
// キューからのデータのデキュー
// front:データの取り出し位置を示すポインタ
// rear:データの格納位置を示すポインタ
dequeue(QUEUE) {// QUEUE:キュー（配列名）
  if (front == rear)
    "Queue Underflow !!" と出力して終了;
  x = QUEUE[front];
  front = front+1;
  return x;
}
```

キューの実装

- リングバッファ
  - frontもrearも単調に増加するので，操作を続けていくうちに，いずれの配列のサイズを超えてしまう．
  - front，rearがn−1に達した場合は，さらに進める際はnではなく0に戻る．

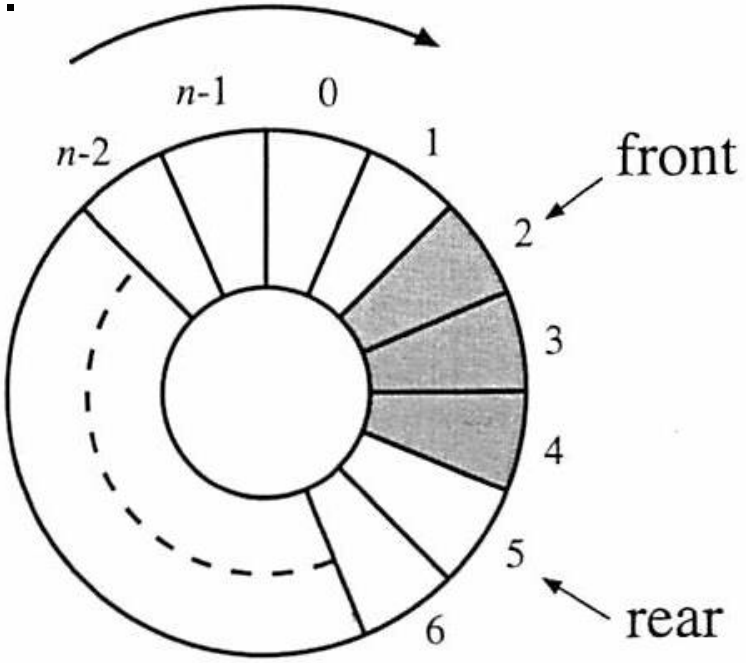


図 2.15　リングバッファ

キューの実装

- リングバッファ
  - 問題：front=rearの場合に，キューが空なのか満杯なのか区別できない．
    解決策：rearを1つ進めた際にrear=frontとなる場合は，オーバーフローとしてエンキューしない．下記の図の状態を満杯状態とみなす．

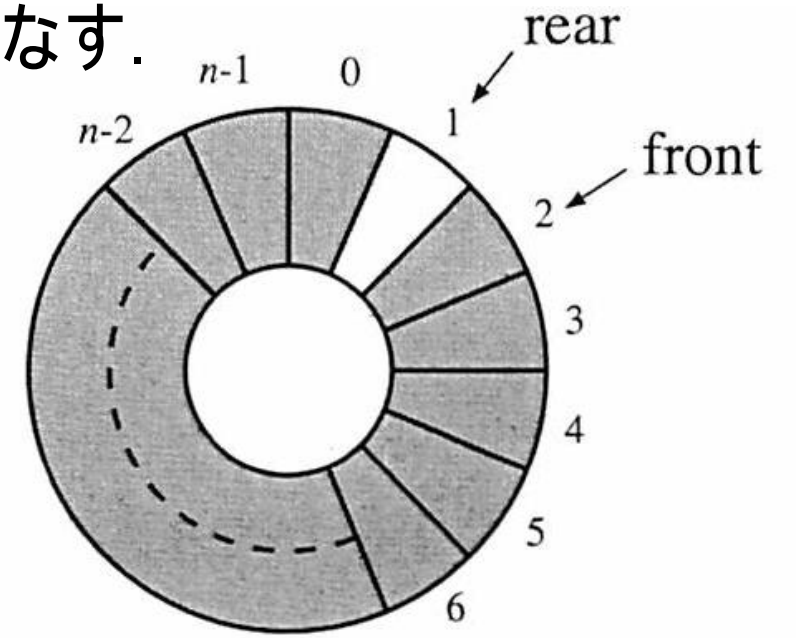


図 2.16　空き要素が 1 つのリングバッファ

木構造(木，tree)

- 節点(node)，辺(edge)，ラベル(label)
- 子(child)，親(parent)，兄弟(sibling)
- 根(root)，葉(leaf)
- 終端節点(terminal node)，非終端節点(non-terminal node)

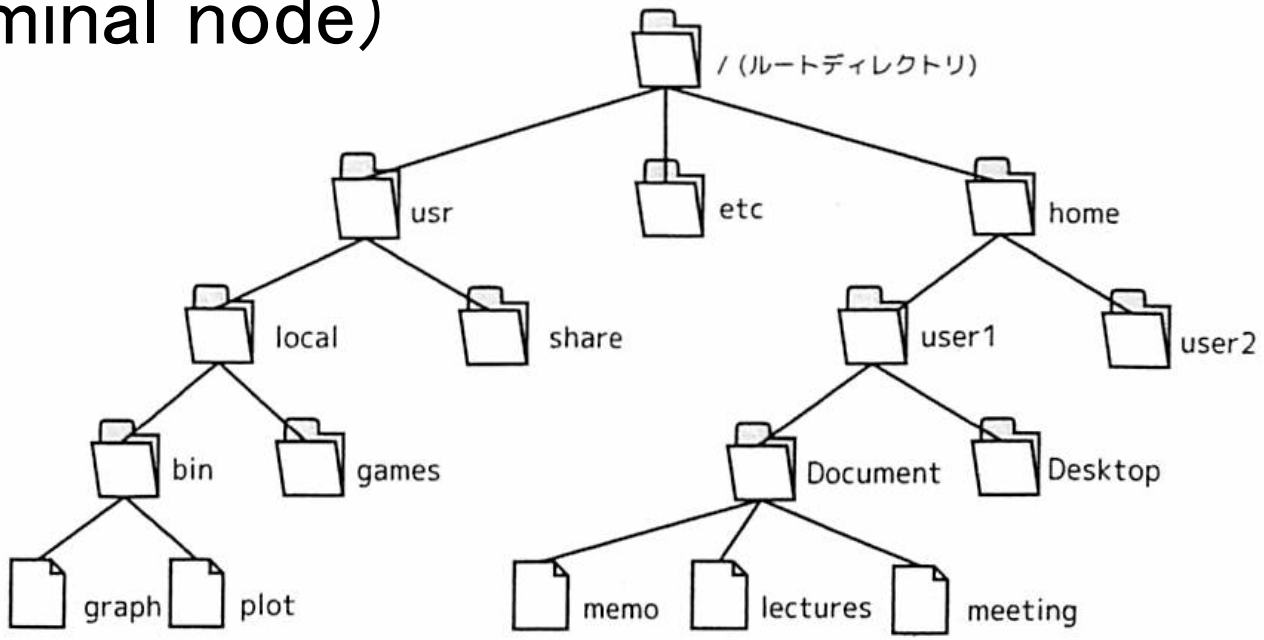


図 2.17 ディレクトリツリー

木構造(木，tree)

- レベル(level)，深さ(depth)
- 木の高さ：最大のレベル
- 部分木(subtree)

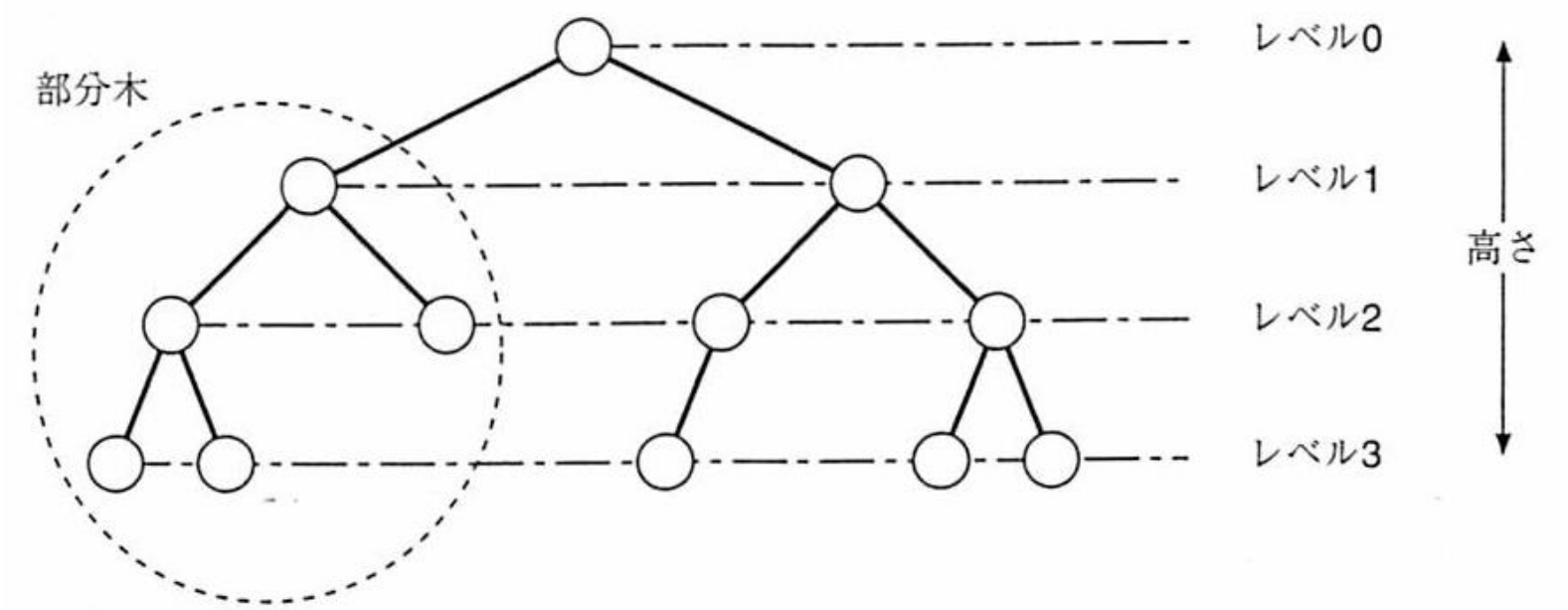


図 2.18　木のレベル

2分木(binary tree)

- k分木：全ての節点がたかだかk個の子をもつ木
- 2分木：k=2の場合
- 左の子，右の子：たとえ子を1個しかもたない場合でも，必ず左の子か右の子か指定する．
- 左部分木，右部分木
- 一般に兄弟の順序に意味がある木を順序木(ordered tree)という．

2分木ではない

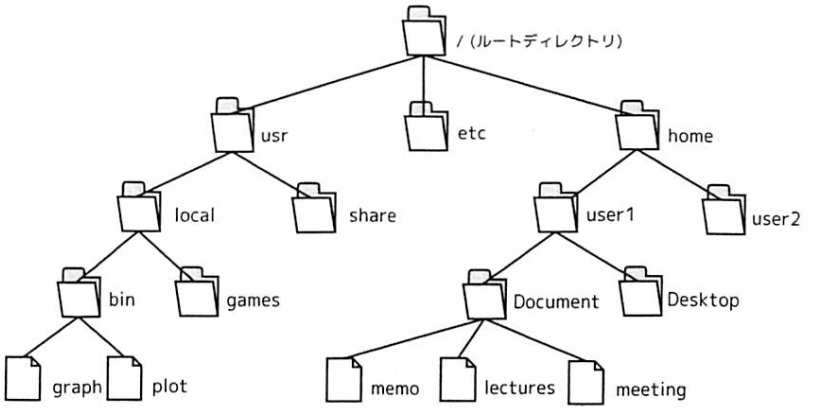


図 2.17 ディレクトリツリー

2分木

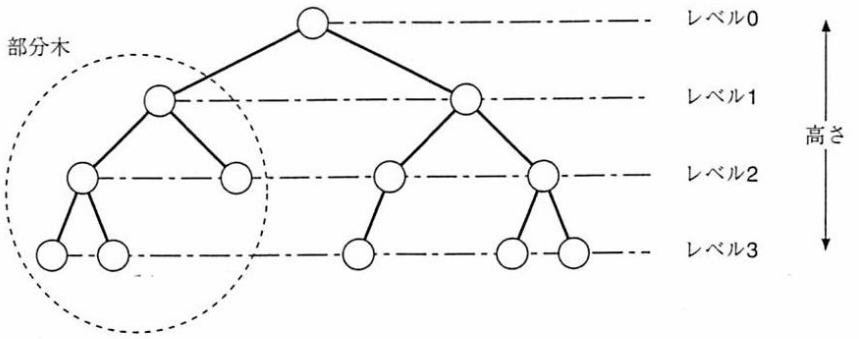


図 2.18 木のレベル

完全2分木

- 最大レベルを除いたどのレベルも完全に節点が詰まっており，かつ最大レベルでは節点が左詰めになっている2分木.
- 狭義には，最大レベルも完全に節点がつまっている木.

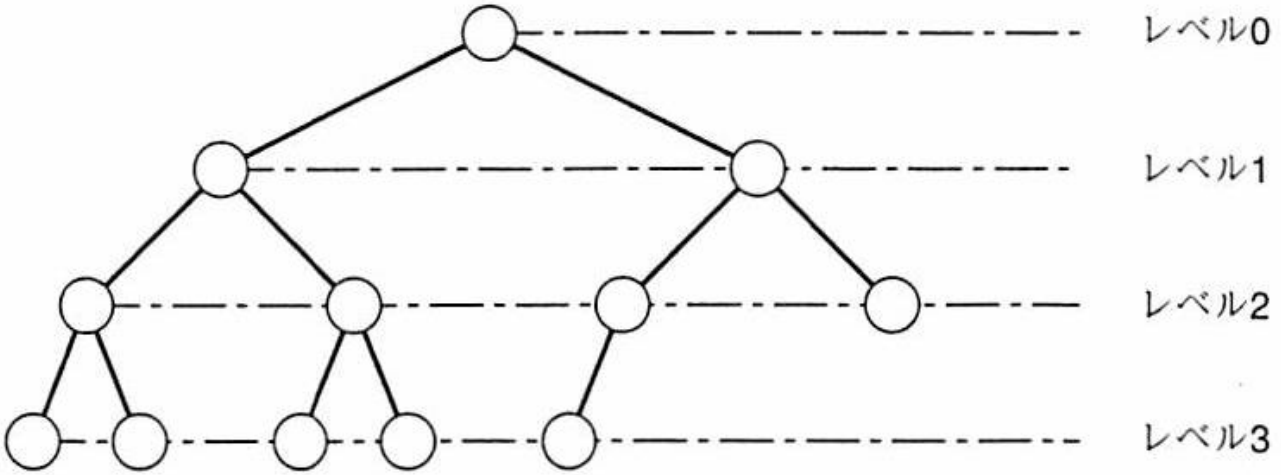


図 2.19　完全 2 分木（広義）

- 高さhの完全2分木がもつ節点数：$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 = O(2^h)$
- 根から葉までたどる処理の時間計算量：O(log(n))

リンク配置による2分木の実現

- セル：ラベル部，左の子へのポインタ，右の子へのポインタ
- 対応する子がない場合は，ポインタ部にはNULL（斜線）が入る．

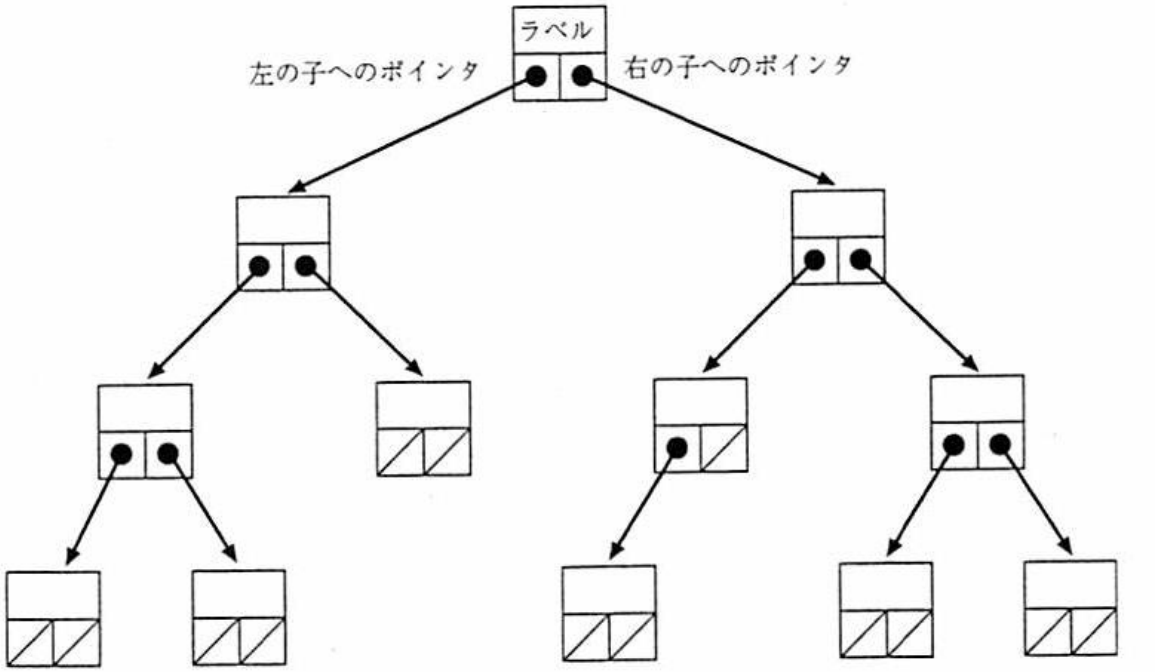


図 2.20　リンク配置による 2 分木の実現

配列を用いた完全2分木の実現

- 完全2分木は配列を用いて効率的に実現することが可能.

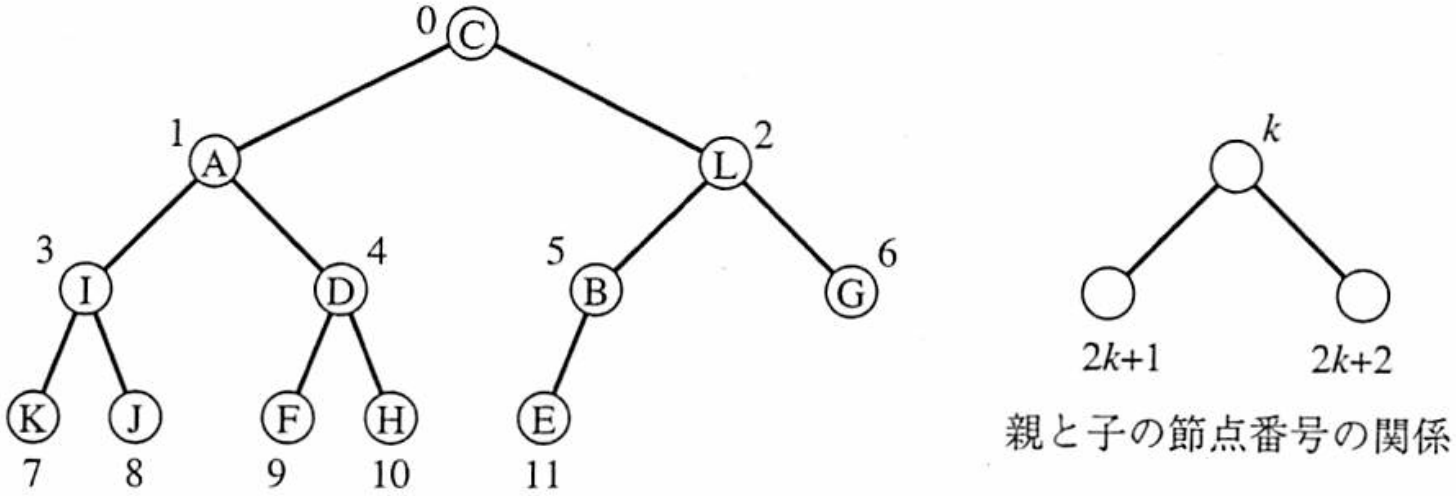


図 2.21　完全 2 分木の番号付け

	0	1	2	3	4	5	6	7	8	9	10	11
TREE	C	A	L	I	D	B	G	K	J	F	H	E

図 2.22　配列に格納した完全 2 分木のラベルの様子

木の走査

- 走査(traverse)：木の全ての節点を見落としや重複なく系統的に訪れること.
- 深さ優先探索（縦型探索，depth first search）：根から始めて葉に至るまでの一本の経路をたどった後に次の経路に移る．木のなぞり.
- 幅優先探索（横型探索，breadth first search）：根から始めてレベルの小さい順に同一レベルの節点を左からの順にたどった後に次のレベルに移る.

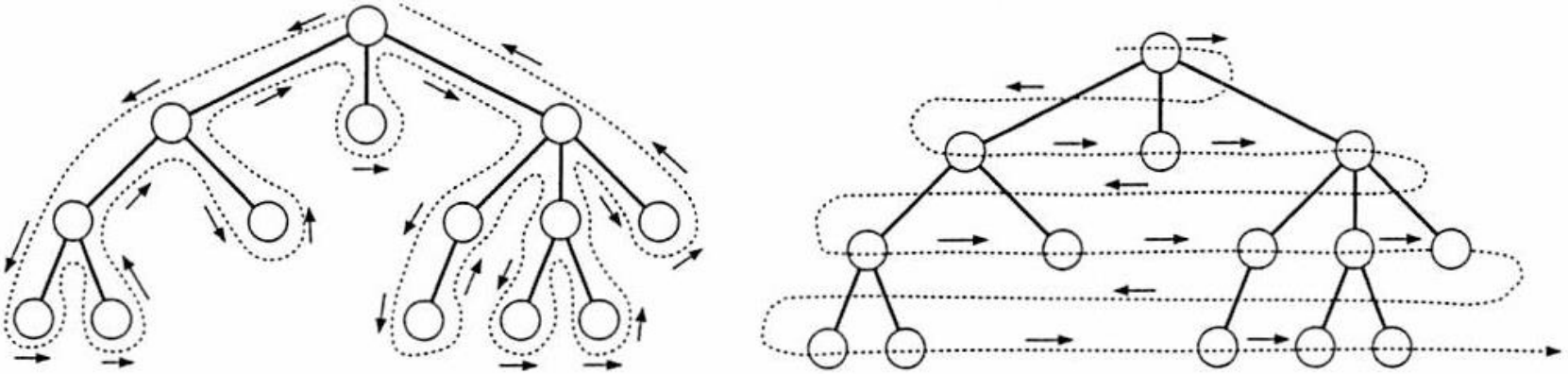


図 2.23　深さ優先探索と幅優先探索

深さ優先探索

- 行きがけ順（先行順，preorder）
  - 根に立ち寄る
  - 左部分木を行きがけ順でなぞる
  - 右部分木を行きがけ順でなぞる
- 通りがけ順（中間順，inorder）
  - 左部分木を通りがけ順でなぞる
  - 根に立ち寄る
  - 右部分木を通りがけ順でなぞる
- 帰りがけ順（後行順，postorder）
  - 左部分木を帰りがけ順でなぞる
  - 右部分木を帰りがけ順でなぞる
  - 根に立ち寄る

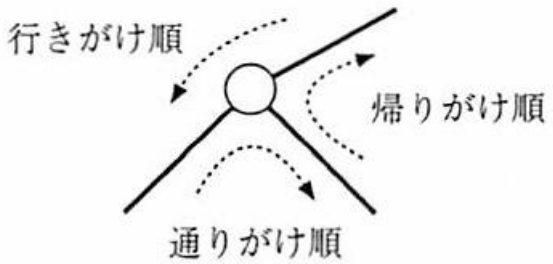
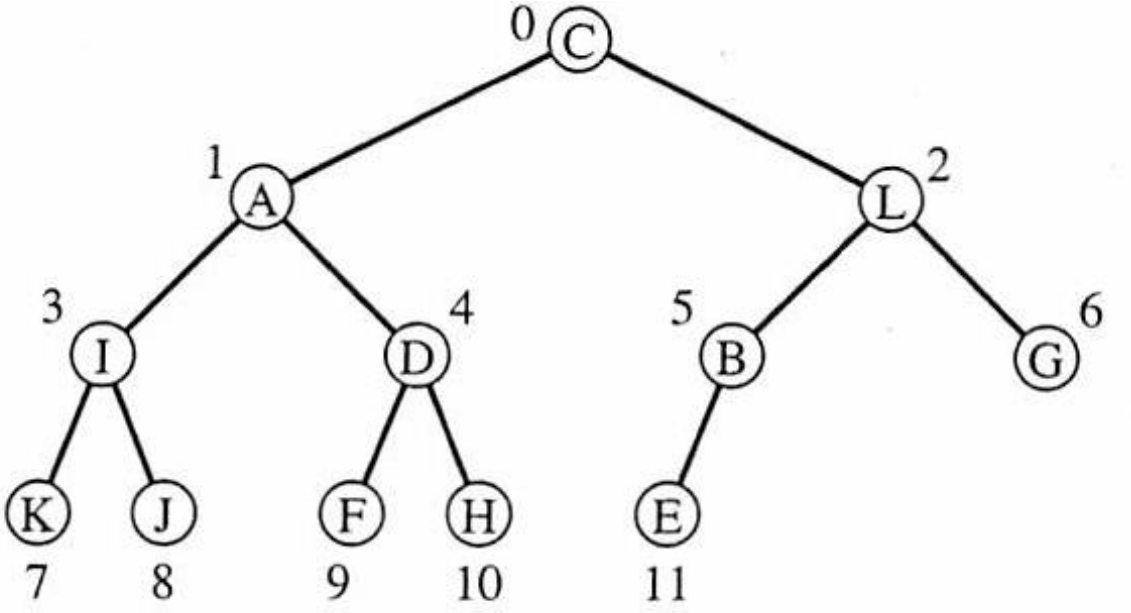


図 2.24　深さ優先探索のデータ処理のタイミング

深さ優先探索

- 行きがけ順（先行順，preorder）
  - C→A→I→K→J→D→F→H→L→B→E→G
- 通りがけ順（中間順，inorder）
  - K→I→J→A→F→D→H→C→E→B→L→G
- 帰りがけ順（後行順，postorder）
  - K→J→I→F→H→D→A→E→B→G→L→C



深さ優先探索

リスト 2.11 木のなぞり

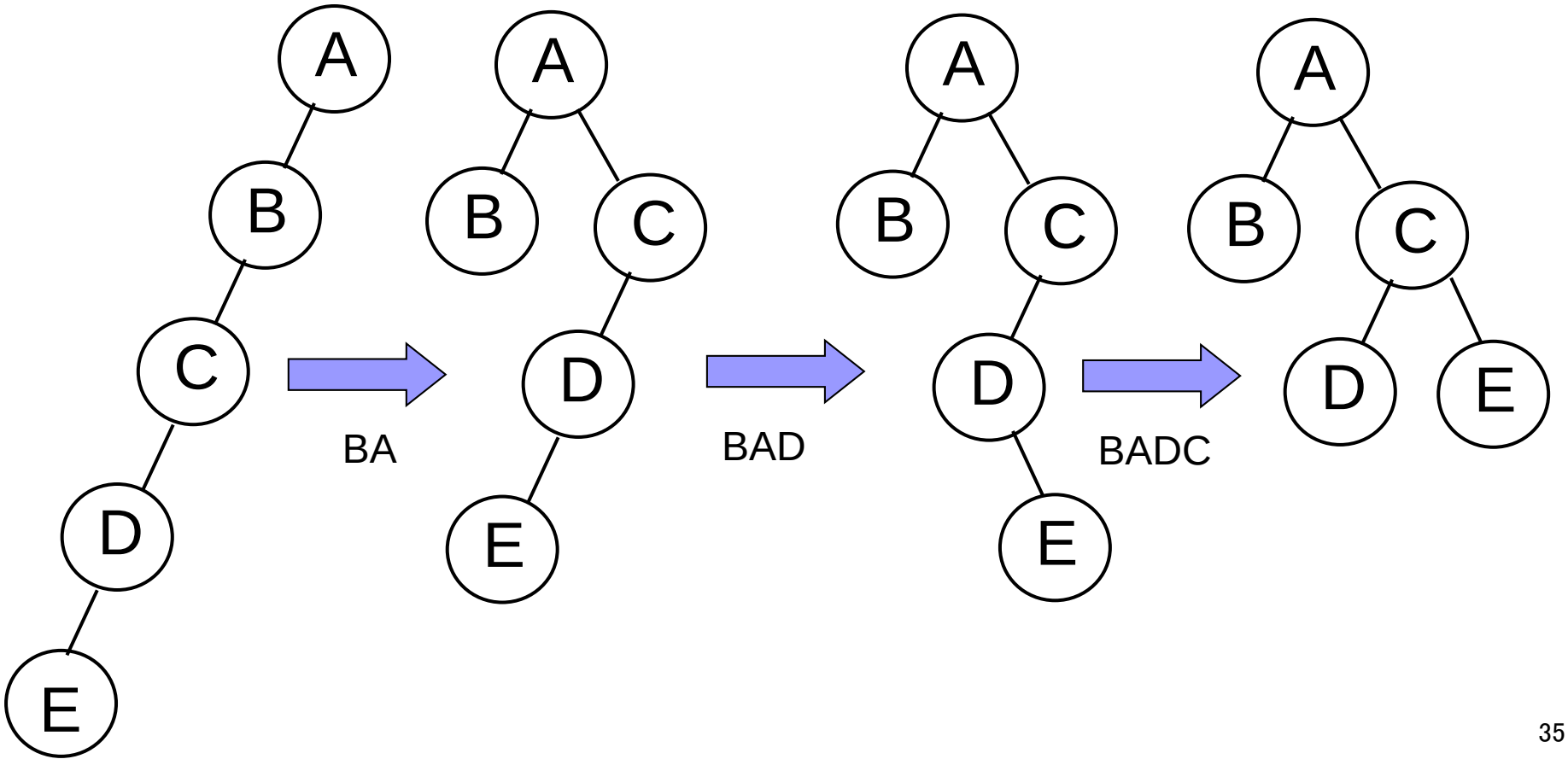
```
// node を根とする部分木の行きがけ順のなぞり
preorder(node) {
  if (node == NULL) return; // 節点が存在しない場合何もせず復帰
  node が指す節点のラベルを出力;
  preorder(node が指す節点の左の子へのポインタ);
  preorder(node が指す節点の右の子へのポインタ);
}

// node を根とする部分木の通りがけ順のなぞり
inorder(node) {
  if (node == NULL) return; // 節点が存在しない場合何もせず復帰
  inorder(node が指す節点の左の子へのポインタ);
  node が指す節点のラベルを出力;
  inorder(node が指す節点の右の子へのポインタ);
}

// node を根とする部分木の帰りがけ順のなぞり
postorder(node) {
  if (node == NULL) return; // 節点が存在しない場合何もせず復帰
  postorder(node が指す節点の左の子へのポインタ);
  postorder(node が指す節点の右の子へのポインタ);
  node が指す節点のラベルを出力;
}
```

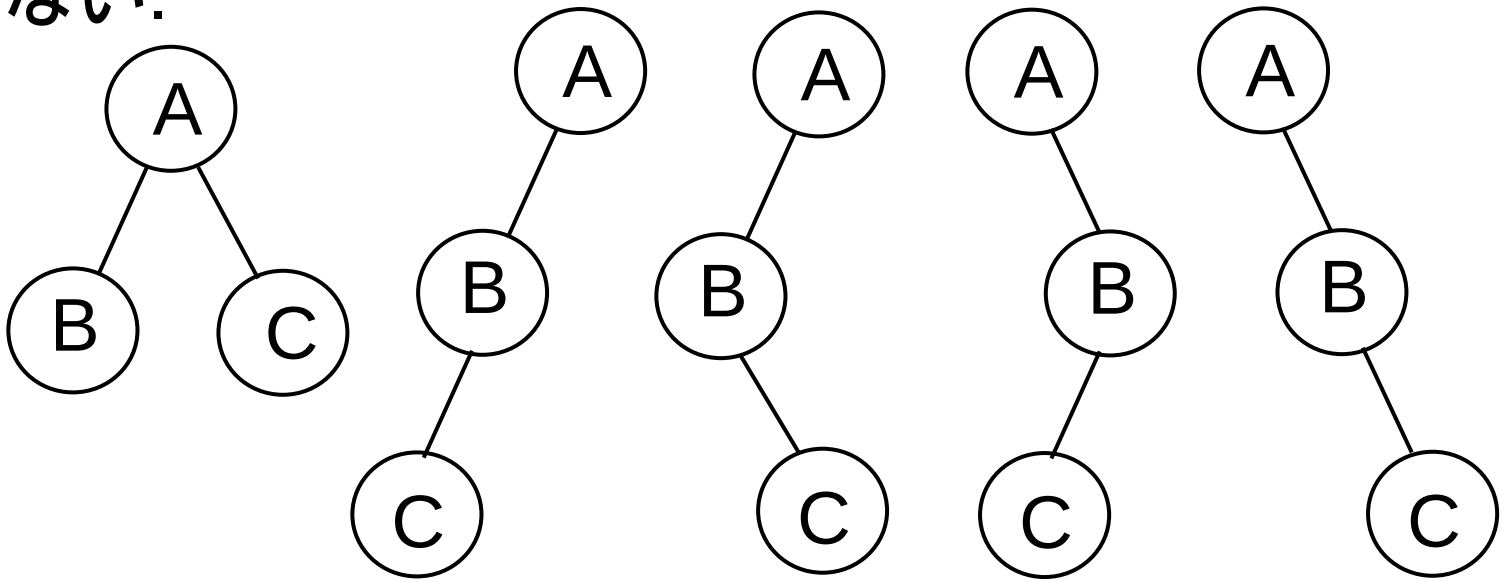
走査からの二分木の決定

- 行きがけ順と通りがけ順，通りがけ順と帰りがけ順から二分木の構造は決定可能.
- 例　行きがけ順：ABCDE，通りがけ順：BADCE



走査からの二分木の決定

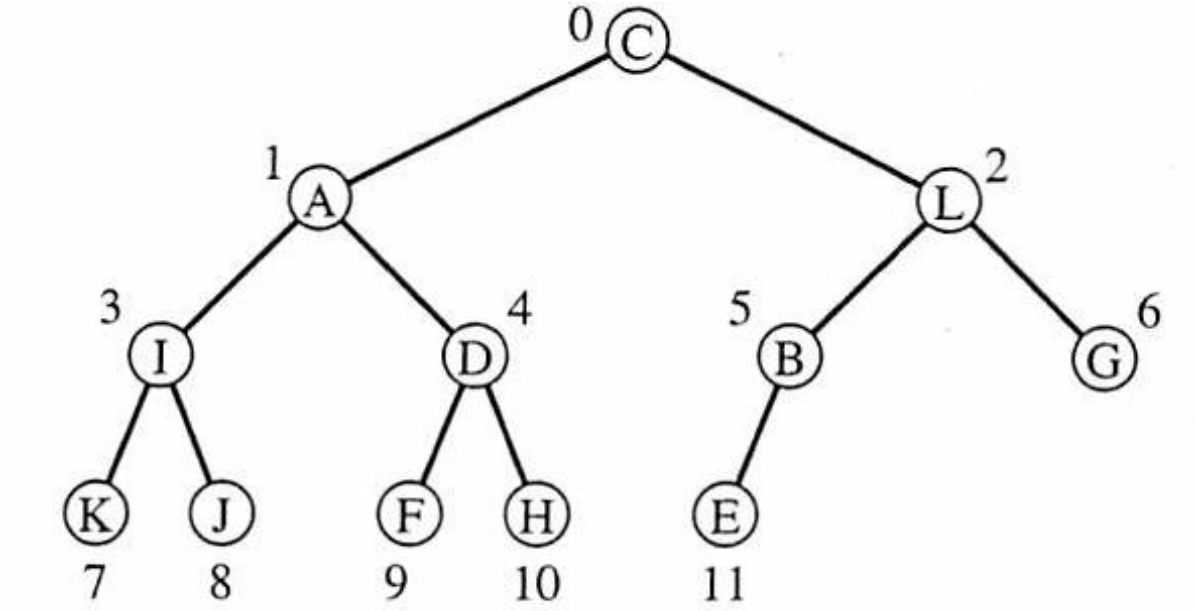
- 行きがけ順と帰りがけ順から二分木の構造は決定できない.



- 行きがけ順：ABC　ABC　ABC　ABC　ABC
- 通りがけ順：BAC　CBA　BCA　ACB　ABC
- 帰りがけ順：BCA　CBA　CBA　CBA　CBA

幅優先探索

- C→A→L→I→D→B→G→K→J→F→H→E



幅優先探索

リスト 2.12 木の幅優先探索

```
// 木の幅優先探索
// QUEUE:キュー, root:根へのポインタ
breadth_first_search() {
  enqueue(QUEUE, root);
  while (QUEUE が空でない) {
    節点へのポインタ node = dequeue(QUEUE);
    node が指す節点のラベルを出力;
    if (node が指す節点の左の子が存在する)
      enqueue(QUEUE, node が指す節点の左の子へのポインタ);
    if (node が指す節点の右の子が存在する)
      enqueue(QUEUE, node が指す節点の右の子へのポインタ);
  }
}
```

一般の木の実現

- 子をつなぐ連結リストを用いる方法
  - 節点を表すセル：ラベル，子をつなぐ連結リストの先頭セルへのポインタ．
  - 子をつなぐ連結リストを作るためのセル：子を表すセルへのポインタ，次のセルへのポインタ．

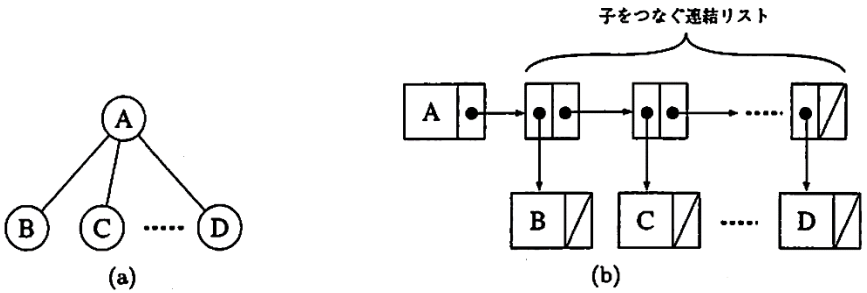


図 2.25 (a) 一般の木の親子の様子と (b) その実現（子をつなぐ連結リストを用いた方法）

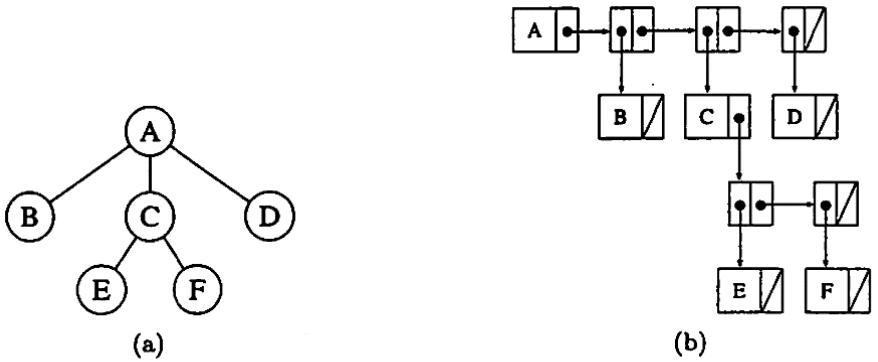


図 2.26 (a) 一般の木と (b) その実現（子をつなぐ連結リストを用いた方法）

一般の木の実現

- 等価な2分木に変換する方法（長男・次弟）
  - セル：ラベル，長男へのポインタ，自分のすぐ右の兄弟（次弟）へのポインタ．

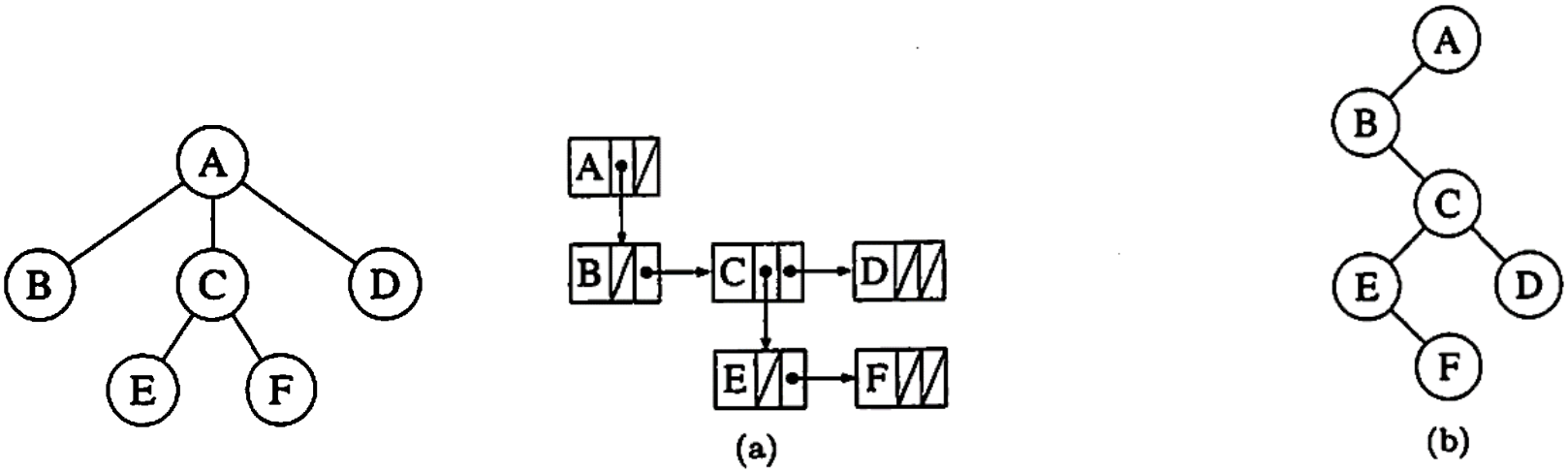


図 2.27　(a) 一般の木の実現（等価な 2 分木への変換による方法）と (b) 等価な 2 分木

一般の木の走査

- 深さ優先探索
  - 行きがけ順（先行順，preorder）
    - 根に立ち寄る
    - 各子を根とする部分木を行きがけ順でなぞる
  - 通りがけ順（中間順，inorder）
    - 最も左の子を根とする部分木を通りがけ順でなぞる
    - 根に立ち寄る
    - ２番目以降の各子を根とする部分木を通りがけ順でなぞる
  - 帰りがけ順（後行順，postorder）
    - 各子を根とする部分木を帰りがけ順でなぞる
    - 根に立ち寄る
- 幅優先探索
  - 節点のラベルを出力した後，全ての子へのポインタをその順にエンキューする．