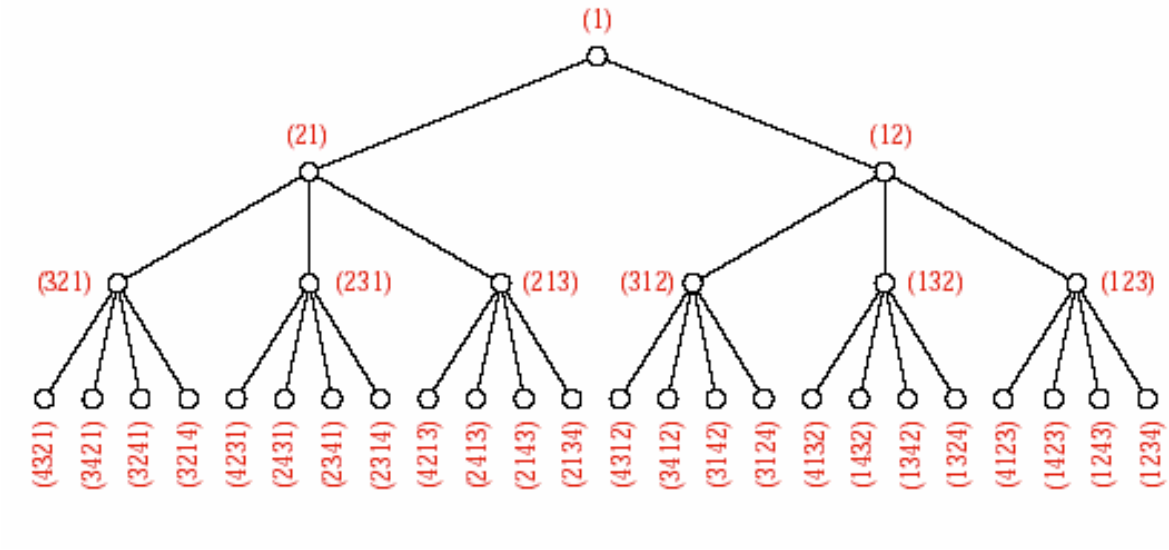
Chương 17 – ỨNG DỤNG SINH CÁC HOÁN VỊ

Ứng dụng này minh họa sự sử dụng cả hai loại danh sách: danh sách tổng quát và DSLK trong mảng liên tục. Ứng dụng này sẽ sinh ra n! cách hoán vị của n đối tượng một cách hiệu quả nhất. Chúng ta gọi các hoán vị của n đối tượng khác nhau là tất cả các phương án thiết lập chúng theo mọi thứ tự có thể có.

Chúng ta có thể chọn bất kỳ đối tượng nào trong n đối tượng đặt tại vị trí đầu tiên, sau đó có thể chọn bất kỳ trong n-1 đối tượng còn lại đặt tại vị trí thứ hai, và cứ thế tiếp tục. Các chọn lựa này độc lập nhau nên tổng số cách chọn lựa là:

```
n x (n-1) x (n-2) x ... x 2 x 1 = n!
```



Hình 17.1- Sinh các hoán vị cho {1, 2, 3, 4}

17.1. Ý tưởng

Chúng ta xác định các hoán vị qua các nút trên cây. Đầu tiên chỉ có 1 ở gốc cây. Chúng ta có hai hoán vị của {1, 2} bằng cách ghi 2 bên trái, sau đó bên phải của 1. Tương tự, sáu hoán vị của {1, 2, 3} có được từ (2, 1) và (1, 2) bằng cách thêm 3 vào cả ba vị trí có thể (trái, giữa, phải). Như vậy các hoán vị của {1, 2, ..., k} có được như sau:

Đối với mỗi hoán vị của {1, 2, ..., k-1} chúng ta đưa các phần tử vào một list. Sau đó chèn k vào mọi vị trí có thể trong list đó để có được k hoán vị khác nhau của {1, 2, ..., k}.

Giải thuật này minh họa việc sử dụng đệ quy để hoàn thành các công việc đã tạm hoãn. Chúng ta sẽ viết một hàm, đầu tiên là thêm 1 vào một danh sách rỗng,

sau đó gọi đệ quy để thêm lần lượt các số khác từ 2 đến n vào danh sách. Lần gọi đệ quy đầu tiên sẽ thêm 2 vào danh sách chỉ chứa có 1, giả sử thêm 2 bên trái của 1, và trì hoãn các khả năng thêm khác (như là thêm 2 bên phải của 1), để gọi đệ quy tiếp. Cuối cùng, lần gọi đệ quy thứ n sẽ thêm n vào danh sách. Bằng cách này, bắt đầu từ một cấu trúc cây, chúng ta phát triển một giải thuật trong đó cây này trở thành một cây đệ quy.

17.2. Tinh chế

Chúng ta sẽ phát triển giải thuật trên cụ thể hơn. Hàm thêm các số từ 1 đến n để sinh `n!` hoán vị sẽ được gọi như sau:

```
permute(1, n)
```

Giả sử đã có k-1 số đã được thêm vào danh sách, hàm sau sẽ thêm các số còn lại từ k đến n vào danh sách:

```
void permute(int k,int n)
/*
pre:  Từ 1 đến k - 1 đã có trong danh sách các hoán vị;
post: Chèn các số nguyên từ k đến n vào danh sách các hoán vị.
*/
{
   for //  với mỗi vị trí trong k vị trí có thể trong danh sách.
   {
       //  Chèn k vào vị trí này.
       if (k == n) process_permutation;
       else permute(k + 1, n);
       //  Lấy k ra khỏi vị trí vừa chèn.
   }
}
```

Khi có được một hoán vị đầy đủ của `{1, 2, ..., n}`, chúng ta có thể in kết quả, hoặc gởi kết quả như là thông số vào cho một bài toán nào khác, đó là nhiệm vụ của hàm `process_permutation`.

17.3. Thủ tục chung

Để chuyển giải thuật thành chương trình C++, chúng ta có các tên biến như sau: danh sách các số nguyên **`permutation`** chứa hoán vị của các số; **`new_entry`**, thay cho k, là số nguyên sẽ được thêm vào; và **`degree`**, thay cho n, là số các phần tử cần hoán vị.

```
void permute(int new_entry, int degree, List<int> &permutation)
/*
pre:  permutation chứa 1 hoán vị với các phần tử từ 1 đến new_entry - 1.
post: Mọi hoán vị của degree phần tử được tạo nên từ hoán vị đã có và sẽ được xử lý trong hàm
      process_permutation.
uses: hàm permute một cách đệ quy, hàm process_permutation, và các hàm của List.
*/
{
   for (int current = 0; current < permutation.size() + 1; current++) {
      permutation.insert(current, new_entry);
      if (new_entry == degree)
         process_permutation(permutation);
      else
         permute(new_entry + 1, degree, permutation);
      permutation.remove(current, new_entry);
   }
}
```

Hàm trên đây có thể sử dụng với bất kỳ hiện thực nào của danh sách mà chúng ta đã làm quen (DSLK sử dụng con trỏ, danh sách liên tục, DSLK trong mảng liên tục). Việc xây dựng ứng dụng đầy đủ để sinh các hoán vị xem như bài tập. Tuy nhiên chúng ta sẽ thấy rằng ưu điểm của DSLK trong mảng liên tục rất thích hợp đối với bài toán này trong phần tiếp theo dưới đây.o3

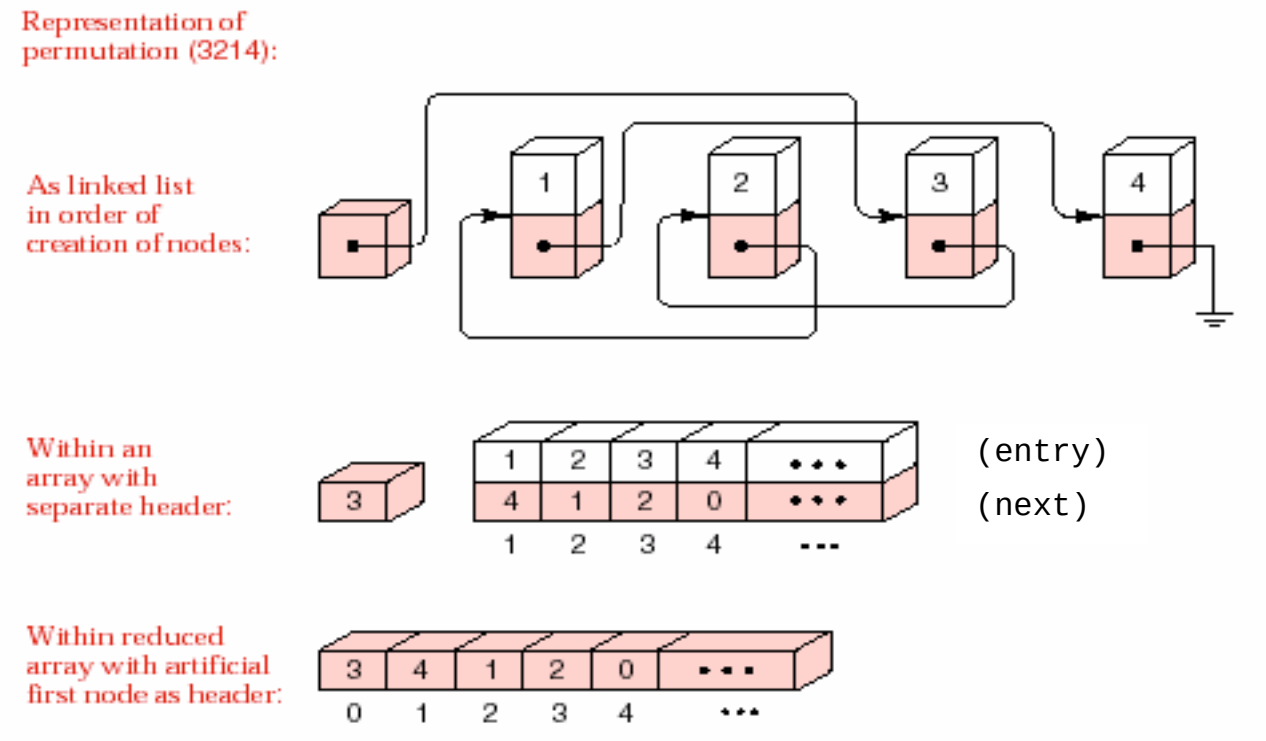
17.4. Tối ưu hóa cấu trúc dữ liệu để tăng tốc độ cho chương trình sinh các hoán vị

n! tăng rất nhanh khi n tăng. Do đó số hoán vị có được sẽ rất lớn. Ứng dụng trên là một trong các ứng dụng mà sự tối ưu hóa để tăng tốc độ chương trình rất đáng để trả giá, ngay cả khi ảnh hưởng đến tính dễ đọc của chương trình. Chúng ta sẽ dùng DSLK trong mảng liên tục có kèm một chút cải tiến cho bài toán trên.

Chúng ta hãy xem xét một vài cách tổ chức dữ liệu theo hướng làm tăng tốc độ chương trình càng nhanh càng tốt. Chúng ta sử dụng một danh sách để chứa các số cần hoán vị. Mỗi lần gọi đệ quy đều phải cập nhật các phần tử trong danh sách. Do chúng ta phải thường xuyên thêm và loại phần tử của danh sách, DSLK tỏ ra thích hợp hơn danh sách liên tục. Mặt khác, do tổng số phần tử trong danh sách không bao giờ vượt quá n, chúng ta nên sử dụng DSLK trong mảng liên tục thay vì DSLK trong bộ nhớ động.

Hình 17.2 minh họa cách tổ chức cấu trúc dữ liệu. Hình trên cùng là DSLK cho hoán vị (3, 2, 1, 4). Hình bên dưới biểu diễn hoán vị này trong DSLK trong mảng liên tục. Đặc biệt trong hình này, chúng ta nhận thấy, trị của phần tử được thêm vào cũng chính bằng chỉ số của phần tử trong array, nên việc lưu các trị này không cần thiết nữa. (Chúng ta chú ý rằng, trong giải thuật đệ quy, các số được thêm vào danh sách theo thứ tự tăng dần, nên mỗi phần tử sẽ chiếm vị trí trong mảng đúng bằng trị của nó; các hoán vị khác nhau của các phần tử này được phân

biệt bởi thứ tự của chúng trong danh sách, đó chính là sự khác nhau về cách nối các tham chiếu). Cuối cùng chỉ còn các tham chiếu là cần lưu trong mảng (hình dưới cùng trong hình 17.2). `Node 0` dùng để chứa đầu vào của DSLK trong mảng liên tục. Trong chương trình dưới đây chúng ta viết lại các công việc thêm và loại phần tử trong danh sách thay vì gọi các phương thức của danh sách để tăng hiệu quả tối đa.



Hình 17.2 – Cấu trúc dữ liệu chứa hoán vị

17.5. Chương trình

Chúng ta có hàm `permute` đã được cải tiến:

```
void permute(int new_entry, int degree, int *permutation)
/*
pre:  permutation chứa 1 hoán vị với các phần tử từ 1 đến new_entry - 1.
post: Mọi hoán vị của degree phần tử được tạo nên từ hoán vị đã có và sẽ được xử lý trong hàm
      process_permutation.
uses: hàm permute một cách đệ quy, hàm process_permutation.
*/
{
   int current = 0;


   do {
      permutation[new_entry] = permutation[current];
      permutation[current] = new_entry;
```

```
      if (new_entry == degree)
         process_permutation(permutation);
      else
         permute(new_entry + 1, degree, permutation);
      permutation[current] = permutation[new_entry];
      current = permutation[current];
   } while (current != 0);
}
```

Chương trình chính thực hiện các khai báo và khởi tạo:

```
main()
/*
pre:  Người sử dụng nhập vào degree là số phần tử cần hoán vị.
post: Mọi hoán vị của degree phần tử được in ra.
*/
{
   int degree;
   int permutation[max_degree + 1];

   cout << "Number of elements to permute? ";
   cin  >> degree;

   if (degree < 1 || degree > max_degree)
      cout << "Number must be between 1 and " << max_degree << endl;
   else {
      permutation[0] = 0;
      permute(1, degree, permutation);
   }
}
```

Danh sách **permutation** làm thông số cho hàm **process_permutation** chứa cách nối kết các phần tử trong một hoán vị, chúng ta có thể in các số nguyên của một cách hoán vị như sau:

```
void process_permutation(int *permutation)
/*
pre:  permutation trong cấu trúc liên kết bởi các chỉ số.
post: permutation được in ra.
*/
{
   int current = 0;
   while (permutation[current] != 0) {
      cout << permutation[current] << " ";
      current = permutation[current];
   }
   cout << endl;
}
```