Cấu trúc dữ liệu và thuật toán

List, Stack và Queue

Danh sách liên kết (List)

- **Các khai báo cần thiết là**

```
typedef ... ElementType; //kiểu phần tử trong danh sách
typedef struct Node
  {
    ElementType Element;//Chứa nội dung của phần tử
    Node* Next; /*con trỏ chỉ đến phần tử  kế tiếp trong
                  danh sách*/
    };
typedef Node* Position; // Kiểu vị trí
typedef Position List;
```

Danh sách liên kết (List)

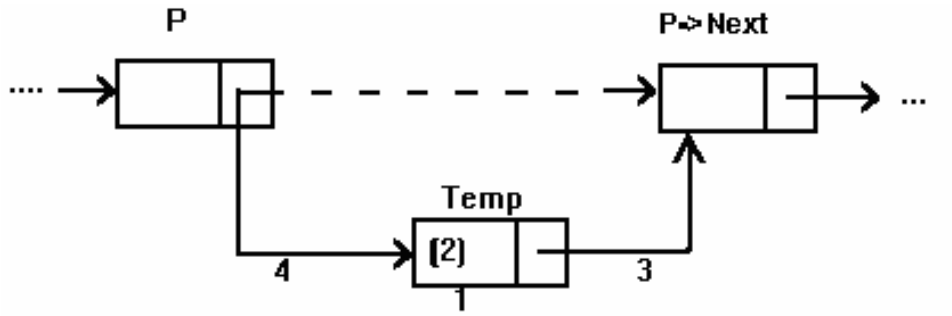
- **Tạo danh sách rỗng:** khi khởi tạo danh sách rỗng, ta phải cấp phát ô nhớ cho HEADER và cho con trỏ trong trường next của nó trỏ tới **NULL**.

```
void MakeNull_List(List *Header)
{
    (*Header)=(Node*)malloc(sizeof(Node));
    (*Header)->Next= NULL;
}
```

- **Kiểm tra danh sách rỗng:** Danh sách rỗng nếu như trường next trong ô Header trỏ tới **NULL**

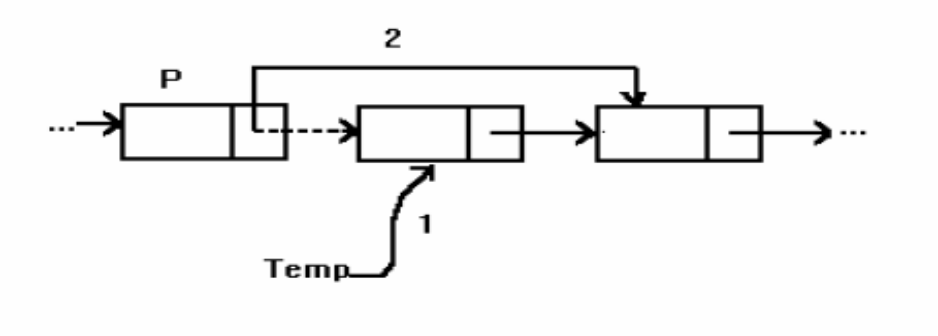
```
int Empty_List(List L)
 {
   return (L->Next==NULL);
  }
```

Danh sách liên kết (List)- chèn




```
// Chèn phần tử vào danh sách tại vị trí p
void Insert_List(ElementType X, Position P, List *L)
{ Position T;
   T=(Node*)malloc(sizeof(Node));
   T->Element=X;
   T->Next=P->Next;
   P->Next=T;
}
```

Danh sách liên kết (List)- Xóa




```
// Xoá phần tử tại vị trí p
void Delete_List(Position P, List *L)
{ Position T;
  if (P->Next!=NULL){
      T=P->Next; /*/giữ ô chứa phần tử bị xoá
                          để thu hồi vùng nhớ*/
       P->Next=T->Next; /*nối kết con trỏ trỏ tới
                                  phần tử thứ p+1*/
 free(T); //thu hồi vùng nhớ
} }
```

Danh sách liên kết (List)- Định vị

Để định vị phần tử x trong danh sách L ta tiến hành tìm từ đầu danh sách (ô header) nếu tìm thấy thì vị trí của phần tử đầu tiên được tìm thấy sẽ được trả về nếu không thì ENDLIST(L) được trả về. Nếu x có trong sách sách và hàm Locate trả về vị trí p mà trong đó ta có

x = p->next->element.

```
Position Locate(ElementType X, List L)
{ Position P;
  int Found = 0;
  P = L;
  while ((P->Next != NULL) && (Found == 0))
       if (P->Next->Element == X) Found = 1;
       else P = P->Next;
  return P;
}
```

Danh sách liên kết (List)- Nội dung ptử

Nội dung phần tử đang lưu trữ tại vị trí p trong danh sách L là p->next->Element Do đó, hàm sẽ trả về giá trị

p->next->element nếu phần tử có tồn tại, ngược lại phần tử không tồn tại (p->next=**NULL**) thì hàm không xác định

```
ElementType Retrieve(Position P, List L)
{
 if (P->Next!=NULL)
   return P->Next->Element;
}
```

Ngăn xếp (Stack)

- **<u>Định nghĩa ngăn xếp</u>**
- *Ngăn xếp (Stack) là một danh sách mà ta giới hạn việc thêm vào hoặc loại bỏ một phần tử chỉ thực hiện tại một đầu của danh sách, đầu này gọi là đỉnh (TOP) của ngăn xếp*
- *Ta có thể xem hình ảnh trực quan của ngăn xếp bằng một chồng đĩa đặt trên bàn. Muốn thêm vào chồng đó 1 đĩa ta để đĩa mới trên đỉnh chồng, muốn lấy các đĩa ra khỏi chồng ta cũng phải lấy đĩa trên trước. Như vậy ngăn xếp là một cấu trúc có tính chất "vào sau - ra trước" hay "vào trước – ra sau"*
- *(**LIFO** (last in - first out) hay **FILO** (first in – last out)).*

Các phép toán trên Ngăn xếp (Stack)

- **MAKENULL_STACK(S):** tạo một ngăn xếp rỗng.
- **TOP(S)** xem như một hàm trả về phần tử tại đỉnh ngăn xếp. Nếu ngăn xếp rỗng thì hàm không xác định. Lưu ý rằng ở đây ta dùng từ "hàm" để ngụ ý là TOP(S) có trả kết quả ra. Nó có thể không đồng nhất với khái niệm hàm trong ngôn ngữ lập trình như C chẳng hạn, vì có thể kiểu phần tử không thể là kiểu kết quả ra của hàm trong C.
- **POP(S)** chương trình con xoá một phần tử tại đỉnh ngăn xếp.
- **PUSH(x,S)** chương trình con thêm một phần tử x vào đầu ngăn xếp.
- **EMPTY_STACK(S)** kiểm tra ngăn xếp rỗng. Hàm cho kết quả 1 (true) nếu ngăn xếp rỗng và 0 (false) trong trường hợp ngược lại.

Ví dụ: Viết chương trình con Edit nhận một chuỗi kí tự từ bàn phím cho đến khi gặp kí tự @ thì kết thúc việc nhập và in kết quả theo thứ tự ngược lại.

```
void Edit(){
 Stack S;
 char c;
 MakeNull_Stack(&S);
do{// Lưu từng ký tự vào ngăn xếp
     c=getche();
     Push(c,&S);
   }while (c!='@');
printf("\nChuoi theo thu tu nguoc lai\n");
  // In ngan xep
while (!Empty_Stack(S)){
    printf("%c\n",Top(S));
   Pop(&S);
   } }
```

Cài đặt ngăn xếp:

- **Cài đặt ngăn xếp bằng danh sách**
- **Khai báo ngăn xếp:**

  typedef List Stack;

- **Tạo ngăn xếp rỗng:**

```
void MakeNull_Stack(Stack *S)
{
   MakeNull_List(S);
 }
```

- **Kiểm tra ngăn xếp rỗng:**

```
int Empty_Stack(Stack S){
    return Empty_List(S);
   }
```

Cài đặt ngăn xếp bằng danh sách

- **Thêm phần tử vào ngăn xếp**

```
void Push(Elementtype X, Stack *S)
{
    Insert_List (x, First (*S), &S);
 }
```

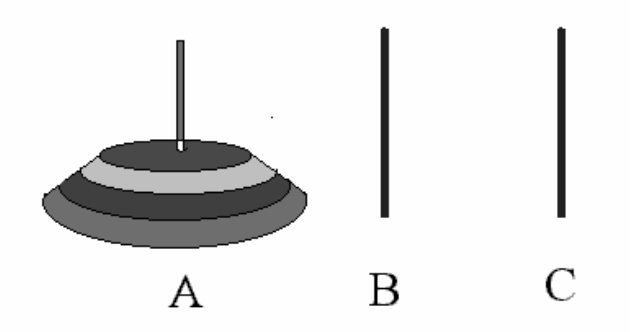
- **Xóa phần tử ra khỏi ngăn xếp**

```
void Pop (Stack *S)
{
   Delete_List (First (*S), &S);
 }
```

Ứng dụng Stack để khử đệ qui

- Ví dụ sau đây minh hoạ việc dùng ngăn xếp để loại bỏ chương trình đệ qui của bài toán "tháp Hà Nội" (tower of Hanoi). Bài toán "tháp Hà Nội" được phát biểu như sau:

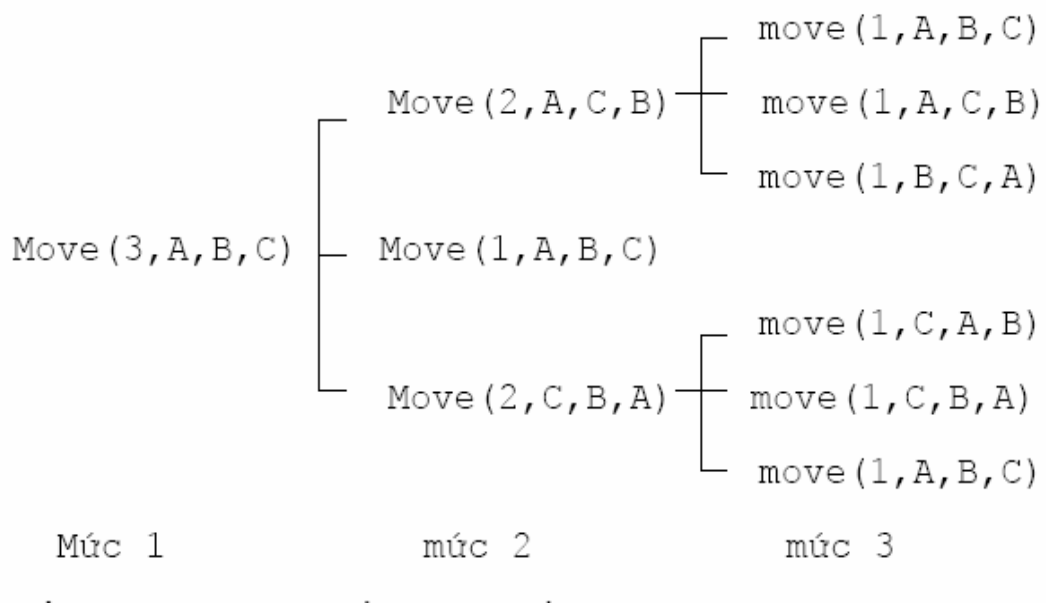
  Có ba cọc A,B,C. Khởi đầu cọc A có một số đĩa xếp theo thứ tự nhỏ dần lên trên đỉnh. Bài toán đặt ra là phải chuyển toàn bộ chồng đĩa từ A sang B. Mỗi lần thực hiện chuyển một đĩa từ một cọc sang một cọc khác và không được đặt đĩa lớn nằm trên đĩa nhỏ



Chương trình con đệ qui cho BT Tháp hà nội

```
void Move(int N, int A, int B, int C)
  //n: số đĩa, A,B,C: cọc nguồn , đích và trung gian
{
  if (n==1)
   printf("Chuyen 1 dia tu %c  sang %c\n",Temp.A,Temp.B);
  else {
     Move(n-1, A,C,B);
     //chuyển n-1 đĩa từ cọc nguồn sang cọc trung gian
     Move(1,A,B,C);
    //chuyển 1 đĩa từ cọc nguồn sang cọc đích
     Move(n-1,C,B,A);
    //chuyển n-1 đĩa từ cọc trung gian sang cọc đích
    }
   }
```

Quá trình thực hiện chương trình con được minh hoạ với ba đĩa (n=3)



Nguyên tắc khử đệ qui

- Mỗi khi chương trình con đệ qui được gọi, ứng với việc đi từ mức i vào mức i+1, ta phải lưu trữ các biến cục bộ của chương trình con ở bước i vào ngăn xếp. Ta cũng phải lưu "địa chỉ mã lệnh" chưa được thi hành của chương trình con ở mức i. Tuy nhiên khi lập trình bằng ngôn ngữ cấp cao thì đây không phải là địa chỉ ô nhớ chứa mã lệnh của máy mà ta sẽ tổ chức sao cho khi mức i+1 hoàn thành thì lệnh tiếp theo sẽ được thực hiện là lệnh đầu tiên chưa được thi hành trong mức i.
- Tập hợp các biến cục bộ của mỗi lần gọi chương trình con xem như là một mẩu tin hoạt động (activation record).
- Mỗi lần thực hiện chương trình con tại mức i thì phải xoá mẩu tin lưu các biến cục bộ ở mức này trong ngăn xếp.
- Như vậy nếu ta tổ chức ngăn xếp hợp lí thì các giá trị trong ngăn xếp chẳng những lưu trữ được các biến cục bộ cho mỗi lần gọi đệ qui, mà còn "điều khiển được thứ tự trở về" của các chương trình con. Ý tưởng này thể hiện trong cài đặt khử đệ qui cho bài toán tháp Hà Nội là: mẩu tin lưu trữ các biến cục bộ của chương trình con thực hiện sau thì được đưa vào ngăn xếp trước để nó được lấy ra dùng sau.

Chương trình con không đệ qui

```
//Kiểu cấu trúc lưu trữ biến cục bộ
typedef struct { int N;  int A, B, C; } ElementType;
// Chương trình con MOVE không đệ qui
void Move(ElementType X){ ElementType Temp, Temp1; Stack S;
    MakeNull_Stack(&S);  Push(X,&S);
 do   {   Temp=Top(S); //Lay phan tu dau
      Pop(&S); //Xoa phan tu dau
      if (Temp.N==1) printf("Chuyen 1 dia tu %c  sang
                        %c\n",Temp.A,Temp.B);     else
   { // Luu cho loi goi Move(n-1,C,B,A)
     Temp1.N=Temp.N-1;     Temp1.A=Temp.C;
     Temp1.B=Temp.B;   Temp1.C=Temp.A;  Push(Temp1,&S);
    // Luu cho loi goi Move(1,A,B,C)
     Temp1.N=1; Temp1.A=Temp.A; Temp1.B=Temp.B;
     Temp1.C=Temp.C;  Push(Temp1,&S);
   //Luu cho loi goi Move(n-1,A,C,B)
   Temp1.N=Temp.N-1; Temp1.A=Temp.A; Temp1.B=Temp.C;
   Temp1.C=Temp.B; Push(Temp1,&S); }
   } while (!Empty_Stack(S));}
```

Minh họa cho lời gọi Move(x) với 3 đĩa, tức là x.N=3.

Ngăn xếp khởi đầu:

3,A,B,C

Ngăn xếp sau lần lặp thứ nhất:

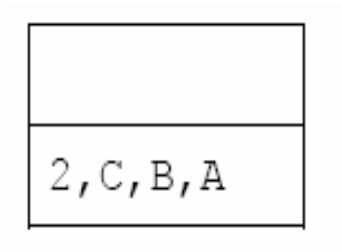
2,A,C,B
1,A,B,C
2,C,B,A

Ngăn xếp sau lần lặp thứ hai

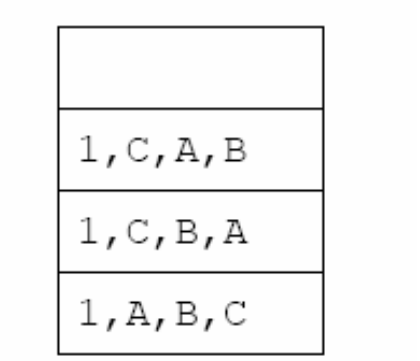
1,A,B,C
1,A,C,B
1,B,C,A
1,A,B,C
2,C,B,A

Minh họa cho lời gọi Move(x) với 3 đĩa, tức là x.N=3.

Các lần lặp 3,4,5,6 thì chương trình con xử lý trường hợp chuyển 1 đĩa (ứng với trường hợp không gọi đệ qui), vì vậy không có mẩu tin nào được thêm vào ngăn xếp. Mỗi lần xử lý, phần tử đầu ngăn xếp bị xoá. Ta sẽ có ngăn xếp như sau.



Tiếp tục lặp bước 7 ta có ngăn xếp như sau:



Các lần lặp tiếp tục chỉ xử lý việc chuyển 1 đĩa (ứng với trường hợp không gọi đệ qui). Chương trình con in ra các phép chuyển và dẫn đến ngăn xếp rỗng.

Hàng đợi QUEUE

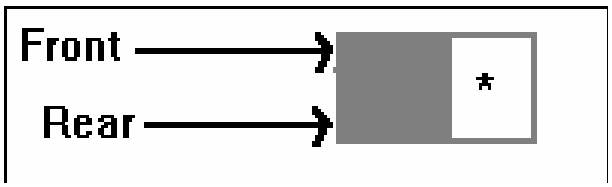
- Hàng đợi, hay ngắn gọn là hàng (queue) cũng là một danh sách đặc biệt mà phép thêm vào chỉ thực hiện tại một đầu của danh sách, gọi là cuối hàng (REAR), còn phép loại bỏ thì thực hiện ở đầu kia của danh sách, gọi là đầu hàng (FRONT).
- Xếp hàng mua vé xem phim là một hình ảnh trực quan của khái niệm trên, người mới đến thêm vào cuối hàng còn người ở đầu hàng mua vé và ra khỏi hang, vì vậy hàng còn được gọi là cấu trúc **FIFO** (first in - first out) hay "vào trước - ra trước".

Các phép toán cơ bản trên hàng

- **MAKENULL_QUEUE(Q)** khởi tạo một hàng rỗng.
- **FRONT(Q)** hàm trả về phần tử đầu tiên của hàng Q.
- **ENQUEUE(x,Q)** thêm phần tử x vào cuối hàng Q.
- **DEQUEUE(Q)** xoá phần tử tại đầu của hàng Q.
- **EMPTY_QUEUE(Q)** hàm kiểm tra hàng rỗng.
- **FULL_QUEUE(Q)** kiểm tra hàng đầy.

Cài đặt hàng bằng danh sách liên kết

- Cách tự nhiên nhất là dùng hai con trỏ front và rear để trỏ tới phần tử đầu và cuối hàng. Hàng được cài đặt như một danh sách liên kết có Header là một ô thực sự



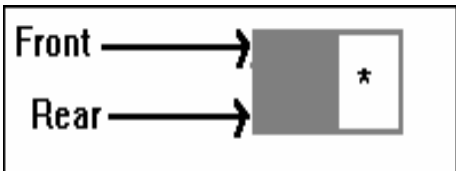
Cài đặt hàng bằng danh sách liên kết

- **Khai báo cần thiết**

```
typedef ... ElementType; //kiểu phần tử của hàng
typedef struct Node{
 ElementType Element;
 Node* Next; //Con trỏ chỉ ô kế tiếp
 };
typedef Node* Position;
typedef struct{
  Position Front, Rear;
   //là hai trường chỉ đến đầu và cuối của hàng
} Queue;
```

Cài đặt hàng bằngDSLK- *Khởi tạo hàng rỗng*

- Khi hàng rỗng Front va Rear cùng trỏ về 1 vị trí đó chính là ô header




```
void MakeNullQueue(Queue *Q){
 Position Header;
 Header=(Node*)malloc(sizeof(Node));
  //Cấp phát Header
 Header->Next=NULL;
 Q->Front=Header;
 Q->Rear=Header;
}
```

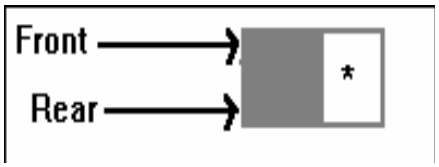
Cài đặt hàng bằngDSLK- *Kiểm tra hàng rỗng*

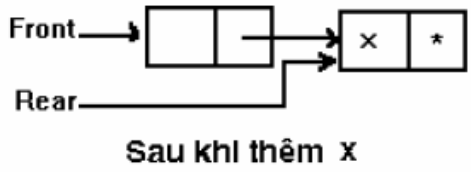
- Hàng rỗng nếu Front và Rear chỉ cùng một vị trí là ô Header.

```
int EmptyQueue(Queue Q)
{
   return (Q.Front==Q.Rear);
 }
```

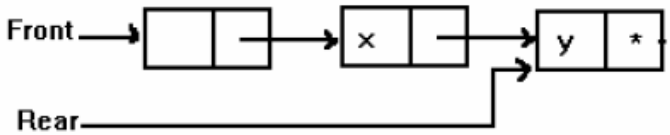
Cài đặt hàng bằngDSLK- *Thêm phần tử*

- Thêm một phần tử vào hàng ta thêm vào sau Rear (Rear->next), rồi cho Rear trỏ đến phần tử mới này. Trường next của ô mới này trỏ tới **NULL**.

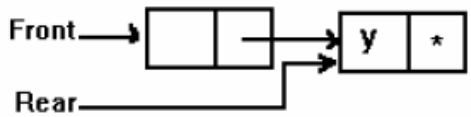




Sau khi thêm x



Sau khi thêm y

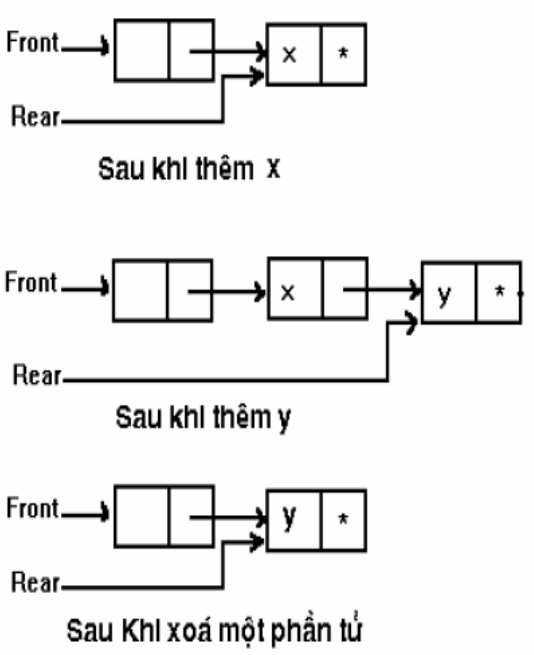


Sau Khi xoá một phần tử

Cài đặt hàng bằngDSLK- *Thêm phần tử*

- Thêm một phần tử vào hàng ta thêm vào sau Rear (Rear->next), rồi cho Rear trỏ đến phần tử mới này. Trường next của ô mới này trỏ tới **NULL**.

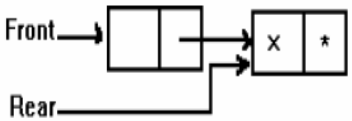
```
void EnQueue(ElementType
X, Queue *Q)
{Q->Rear->Next=
(Node*)malloc(sizeof(Node));
Q->Rear=Q->Rear->Next;
//Dat gia tri vao cho Rear
Q->Rear->Element=X;
Q->Rear->Next=NULL;
}
```



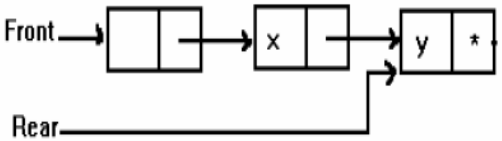
Cài đặt hàng bằngDSLK- *Xóa phần tử*

- Thực chất là xoá phần tử nằm ở vị trí đầu hàng do đó ta chỉ cần cho front trỏ tới vị trí kế tiếp của nó trong hàng.

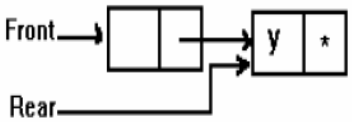
```
void DeQueue(Queue *Q)
{ if (!Empty_Queue(Q)){
    Position T;
    T=Q->Front;
    Q->Front=Q->Front->Next;
    free(T);
     }
 else printf("Loi : Hang rong");
}
```



Sau khi thêm x



Sau khi thêm y



Sau Khi xoá một phần tử

Một số ứng dụng của Hàng đợi QUEUE

- Hàng đợi là một cấu trúc dữ liệu được dùng khá phổ biến trong thiết kế giải thuật. Bất kỳ nơi nào ta cần quản lí dữ liệu, quá trình... theo kiểu vào trước-ra trước đều có thể ứng dụng hàng đợi.
- Ví dụ rất dễ thấy là quản lí in trên mạng, nhiều máy tính yêu cầu in đồng thời và ngay cả một máy tính cũng yêu cầu in nhiều lần. Nói chung có nhiều yêu cầu in dữ liệu, nhưng máy in không thể đáp ứng tức thời tất cả các yêu cầu đó nên chương trình quản lí in sẽ thiết lập một hàng đợi để quản lí các yêu cầu. Yêu cầu nào mà chương trình quản lí in nhận trước nó sẽ giải quyết trước.
- Một ví dụ khác là duyệt cây theo mức được trình bày chi tiết trong chương sau. Các giải thuật duyệt theo chiều rộng một đồ thị có hướng hoặc vô hướng cũng dùng hàng đợi để quản lí các nút đồ thị. Các giải thuật đổi biểu thức trung tố thành hậu tố, tiền tố.