Data structure and Algorithms

LinkedList

Thanh-Hai Tran

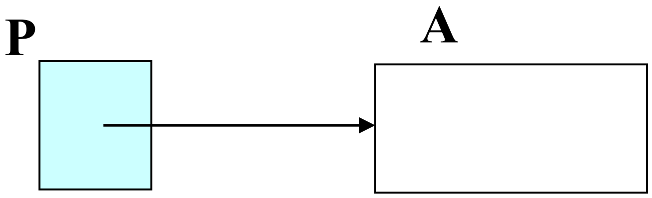
Electronics and Computer Engineering
School of Electronics and Telecommunications

Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

Contents

Introduction

- **Pointers**
  - *Concept of pointer*: a data type used to point to address of object (data object or function object)

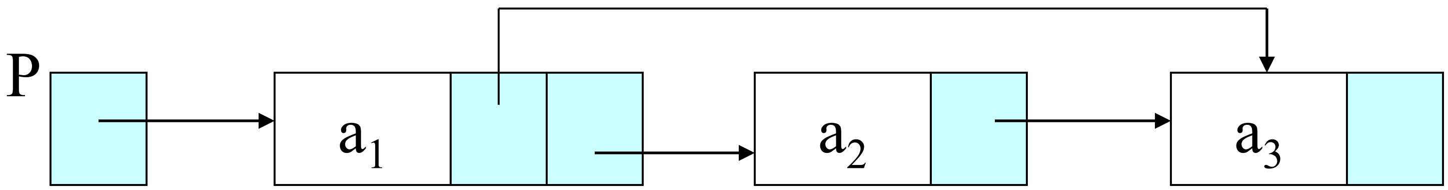


– Basic operations of pointers:

- Declaration: ***int * P;***
- Take address of an object: ***int A; P = &A;***
- Access to pointed object: ****P = 20;***
- Dynamic allocation of memory: ***P = new int;***
- Dynamic deallocation of memory: ***delete P;***

Introduction

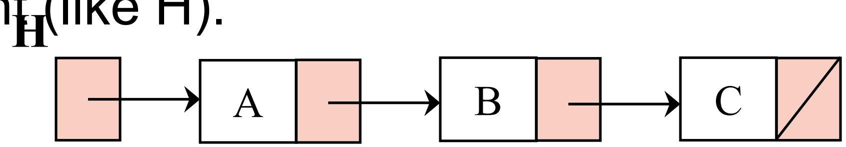
- **Linked Storage Structures**
  - Organization of LSS:



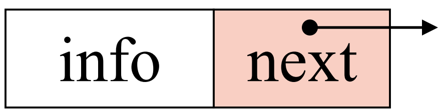
    - **Pointers**: pointing to nodes
    - **Nodes**: each contains information of an element of list and one or more pointers
  - Characteristics of LSS:
    - *Dynamic storage structure*: memory allocation in run-time (on demand)
    - *Flexible arrangement*: pointers can be easily changed to point to different nodes
    - *Must have at least one access point*: where the LSS can be accessed from outside (as shown by P)

Introduction

- **Linked List: list implemented by LSS**
  - *Organization*: 2 components:
    - **Nodes**: each contains information of an element of list and one or more pointers pointing to other nodes.
    - **Pointers**: representing the linear relationships (before-after) among elements. At least one special pointer plays role of access point (like H).



  – *Structure of a node*: 2 parts:
    - **Information**: storing value of an element of list
    - **Next**: Pointer points to next node

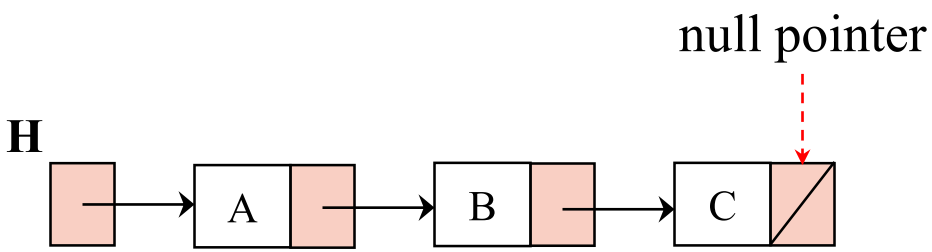


  – **Header**: a pointer (H) points to the first node of the list. It plays role of access point.

Introduction

- **Linked List: list implemented by LSS**
  - Definitions of node and linked list in C:

```
struct  Node {
  Type info;
  struct Node* next;
};
typedef Node*  LinkedList;
```



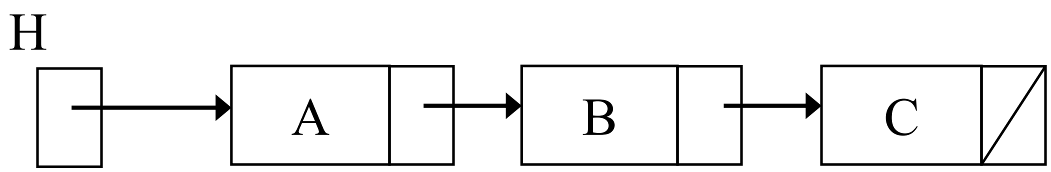
  - *Empty list*:

```
LinkedList H;
H = NULL;
```

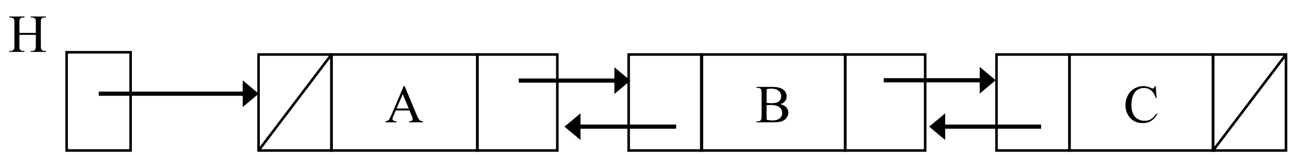
  - *Full list*: When the dynamic memory runs out

Introduction

- Classification of linked lists
  - *By number of pointers in node*
    - *Singly-linked list*:
      - Also called one-way list

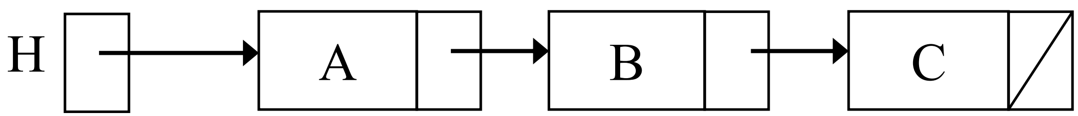


    - *Doubly-linked list:*
      - Also called two-way list

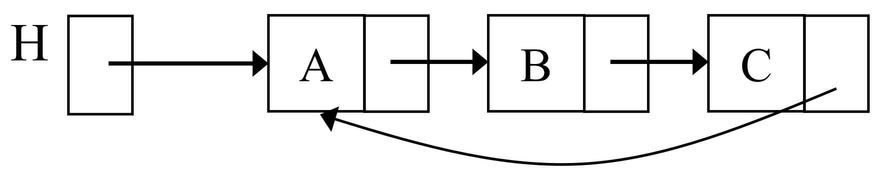


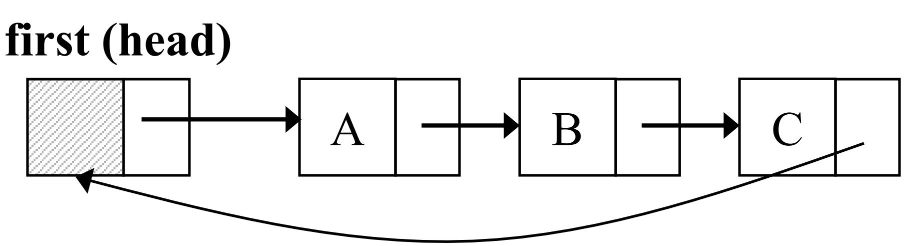
Introduction

- **Classification of linked lists**
  - *By linking ways*:
    - Straightly linked list (Danh sách nối thẳng): having one head node (access point) and one tail node.



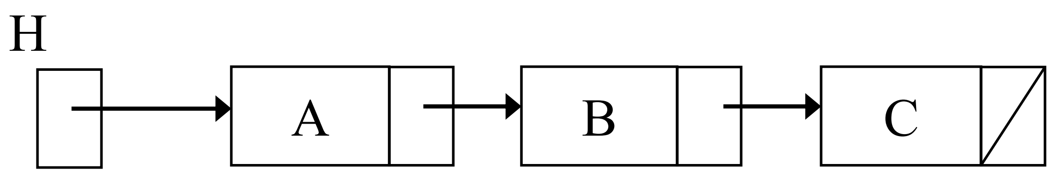
    - *Circularly linked list* (Danh sách nối vòng): every node can be head





Implementation of singly linked list

- **Implementation of straightly linked list (SLL):**
  - Each node has two fields: info and next, next is a pointer pointing to next node.
  - The last node has NULL value for next field.
  - The list has one access point H that points to first node.




```
struct  Node {
   Type info;
   struct Node* next;
};
typedef Node* PNode;
typedef Node* LinkedList;
```

Implementation of singly linked list

- **Implementation of straightly linked list (SLL):**
  - Basic operations:

- Initialize: creating an empty list
- Check current state of list:
  - Empty: when H = NULL
- Insert a new element into list: 2 cases:
  - ***InsertAfter:*** new element inserted after given element
  - ***InsertBefore:*** new element inserted before given element
- Delete an element from list
- Searching for elements
- Traversal of list

Implementation of straightly linked list: Basic operations

- **Initialize:**

```
void Init(LinkedList & H) {
      H = NULL;
}
```

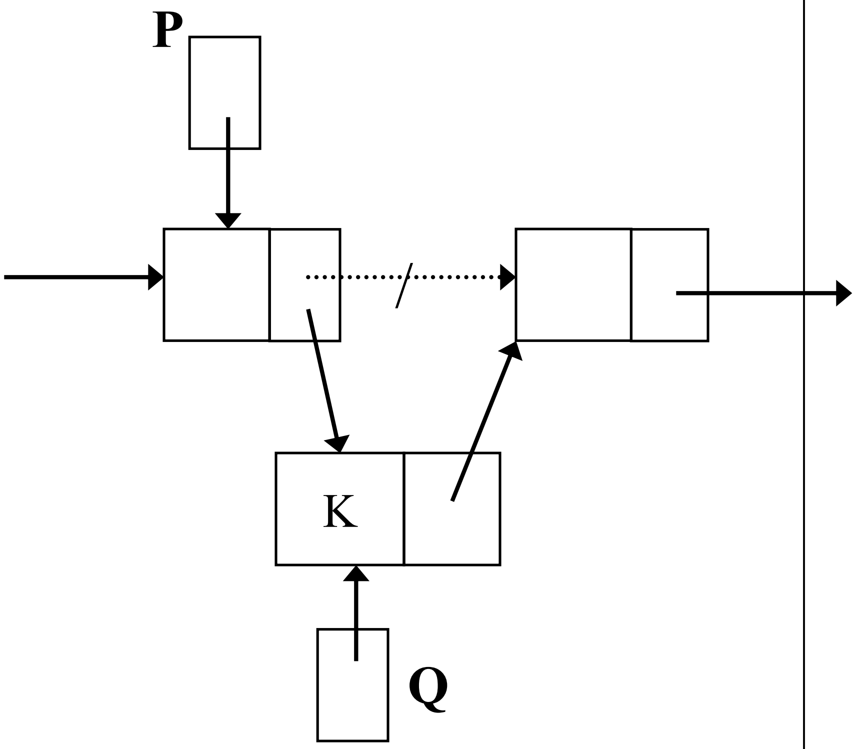
- **Check whether list is empty:**

```
bool IsEmpty (LinkedList  H) {
      return (H == NULL);
}
```

Basic operations

- **InsertAfter: add new element K into list H after given node P. This function returns pointer that points to the new node containing K.**

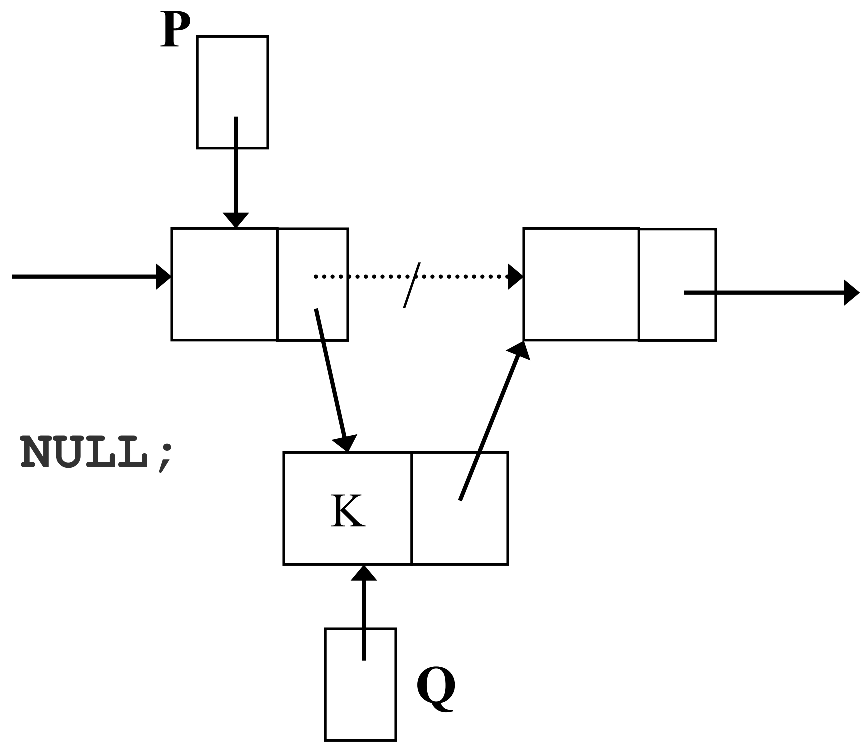
```
PNode InsertAfter(LinkedList & H, PNode P, Type K){
    Allocate new node Q to store K
    If H is empty (H=P=NULL):
        Q->next = NULL;
        H=Q;
    Otherwise:
        Q->next=P->next;
        P->next = Q;
    return Q;
}
```



Source code

```
PNode InsertAfter(LinkedList & H, PNode P, Type K){
    PNode Q = new Node;
    Q->info = K;
    if (H==NULL){
        H = Q;
        Q->next = NULL;

    }else {
        if (P==NULL) return NULL;
        Q->next = P->next;
        P->next = Q;
    }
    return Q;
}
```



Basic operations

- **InsertBefore: add new element K into list H before given node P. This function returns pointer that points to the new node containing K.**

PNode **InsertBefore**(LinkedList & H, PNode P, Type K) {

Allocate new node Q containing K

If H is empty (H=P=NULL):

Q->next = NULL;

H=Q;

Otherwise:

Move info from P to Q

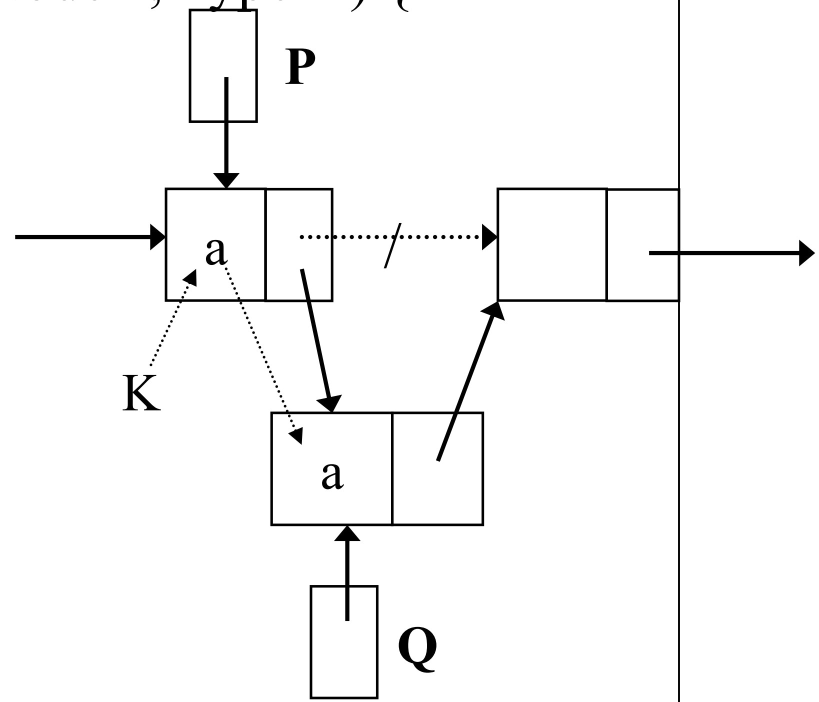
Update info of P by K

Q->next = P->next;

P->next=Q;

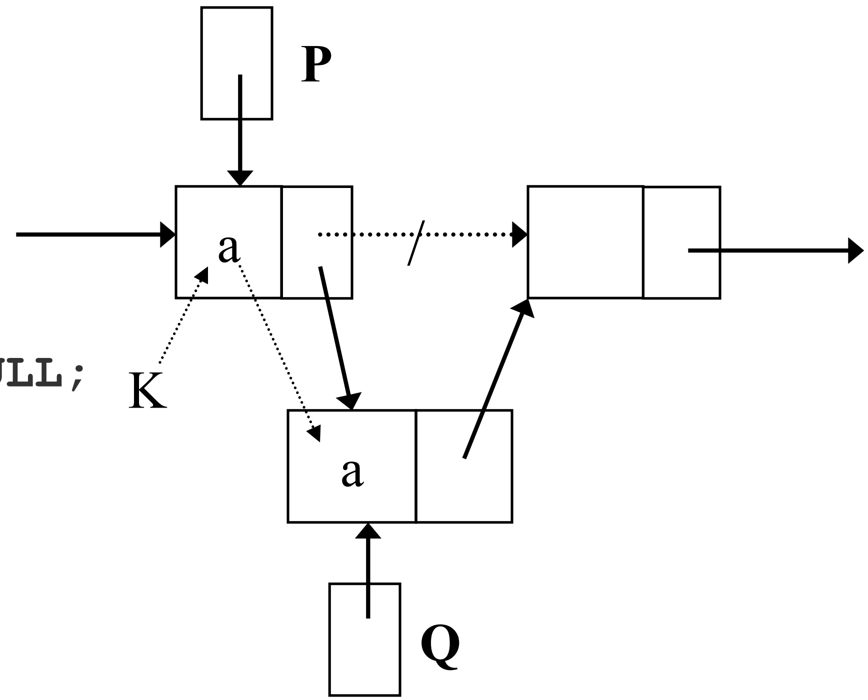
return P;

}



Source code

```
PNode InsertBefore(LinkedList & H, PNode P, Type K){
    PNode Q = new Node;
    Q->info = K;
    if (H==NULL){
        H = Q;
        Q->next = NULL;
        return Q;
    }else {
        if (P==NULL) return NULL;
        Q->info = P->info;
        P->info = K;
        Q->next = P->next;
        P->next = Q;
    }
    return P;
}
```

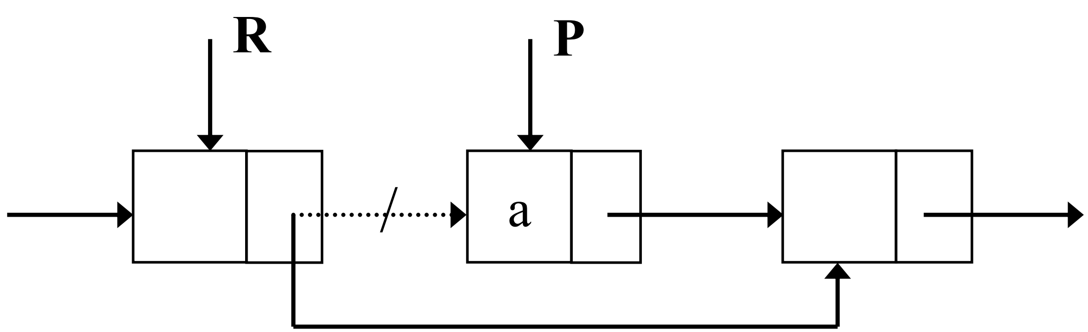


Basic operations

- **Delete a node: deleting the node pointed by P in list H.**

```
void DeleteNode(LinkedList & H, PNode P) {
    If H has only one node (H=P and P->next = NULL)
        Make H empty: H=NULL;
        Release node P: delete P;
    Otherwise
        If (P = H)
            H = H->next;
            Release P
        Otherwise
            Find node R that right before P;
            R->next= P->next;
            Release P;

}
```



Basic operations

- **Function DeleteNode(): it returns the pointer pointing to the next of deleted node;**

```
PNode DeleteNode(LinkedList & H, PNode P){
   if (P==NULL) return NULL;
   if (H==P &&P->next==NULL){//If H has only one node
        H=NULL;
        delete P;
        return NULL;
   }else {
        if (H==P){//If P is the first node
               H=P->next;
               delete P;
               return H;
        }else {
               PNode R=H;
               while (R->next != P) R=R->next;
               R->next = P->next;
               delete P;
               return R->next;
        }
   }
}
```

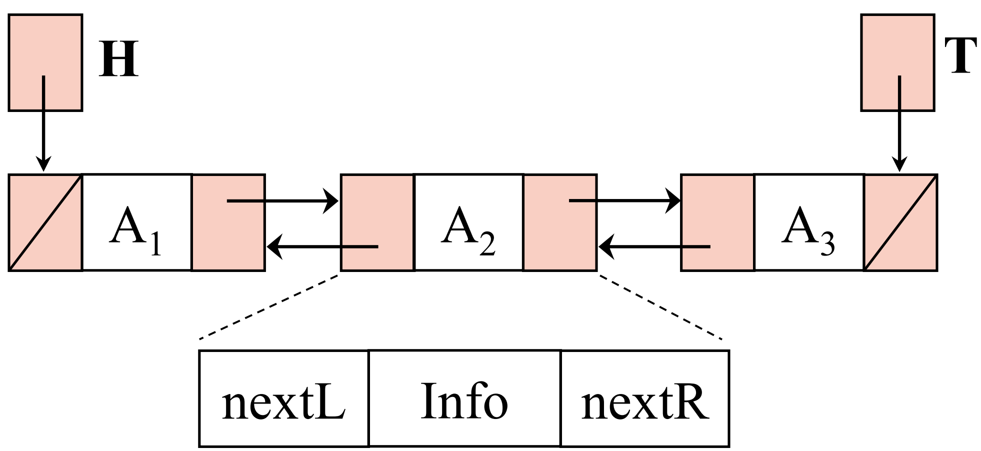
Basic operations

- **Traversal of list: access (visit) to all elements of list one-by-one (maybe used to count the number of list or print list):**

```
void Traverse (LinkedList H) {
   Pnode P;
   P = H;
   while (P != NULL) {
        Visit (P);
        P = P->next;
   }
}
```

Doubly linked lists

- **General organization**
  - A node consists of 3 components:
    - Info:
    - nextL: pointer points to left node (precedent node)
    - nextR: pointer points to right node (following node)
  - One or two pointers play role of access point (as H (head), T (tail))

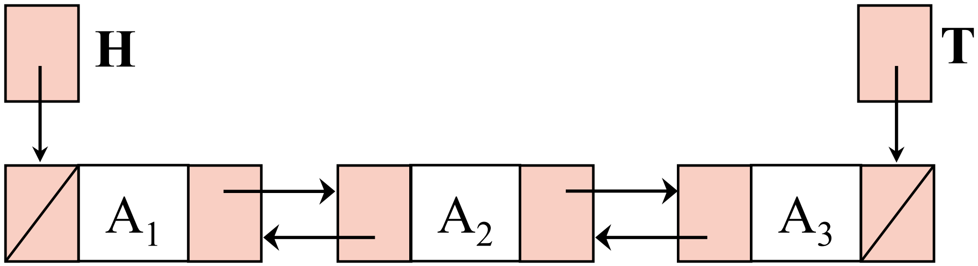


Doubly linked lists

- ## Definition of structure:

```
struct  DNode {
    Type info;
    DNode *nextL, *nextR;
};
typedef DNode*  PDNode;

typedef struct {
    PDNode  H; //head
    PDNode  T; //tail
} DoubleLinkedList;
```



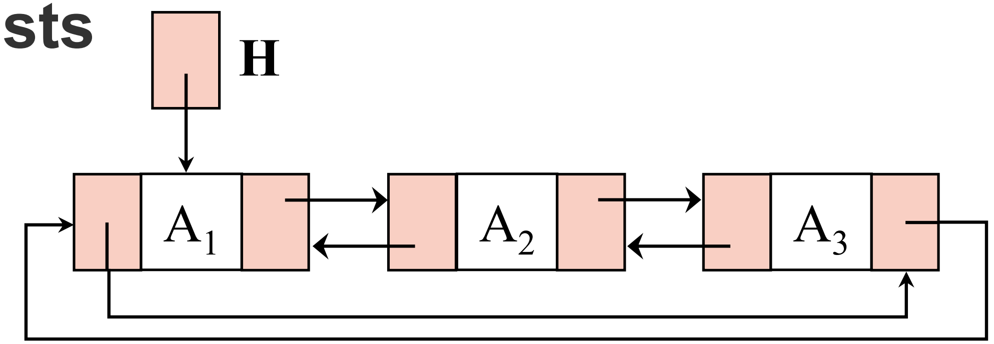
Doubly linked lists

- **Basic Operations:**

  - Initialize: creating an empty list
  - Check current state of list:
    - Empty: when H = T = NULL
  - Insert a new element into list: 2 cases:
    - ***InsertAfter:*** new element inserted after given element
    - ***InsertBefore:*** new element inserted before given element
  - Delete an element from list
  - Searching for elements
  - Traversal of list

Doubly linked lists

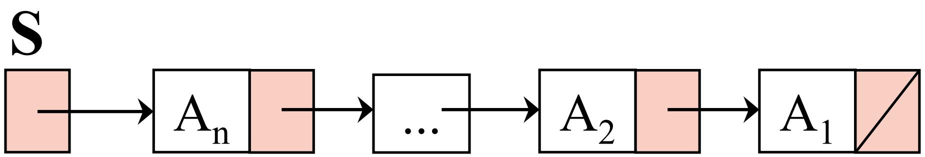
- **Circular doubly linked lists**
  - Empty list:
    - H=NULL




```
struct  DNode {
    Type info;
    DNode *nextL, *nextR;
};
typedef DNode*  PDNode;
typedef PDNode  CDoubleLinkedList;
```

LIFO & FIFO implemented by Linked Storage Structures

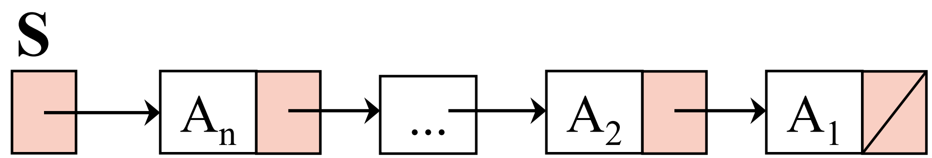
- **LIFO (Stack) implementation:**
  - Organization:
    - Using singly linked list with only one access point S (also top of the list)
  - Definition of structure




```
struct  Node {
   Type info;
   Node* next;
};
typedef Node* PNode;
typedef PNode Stack;
```

LIFO (Stack) implementation

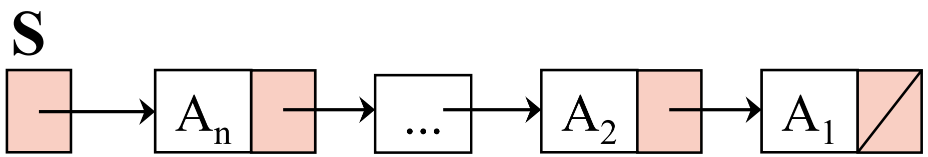
- Operations: similar to linked list operations
  - Initialize
  - isEmpty
  - isFull
  - Push
  - Pop



LIFO (Stack) implementation

- **Operations:**

```
void Initialize (Stack & S) {
   S = NULL;
}
```

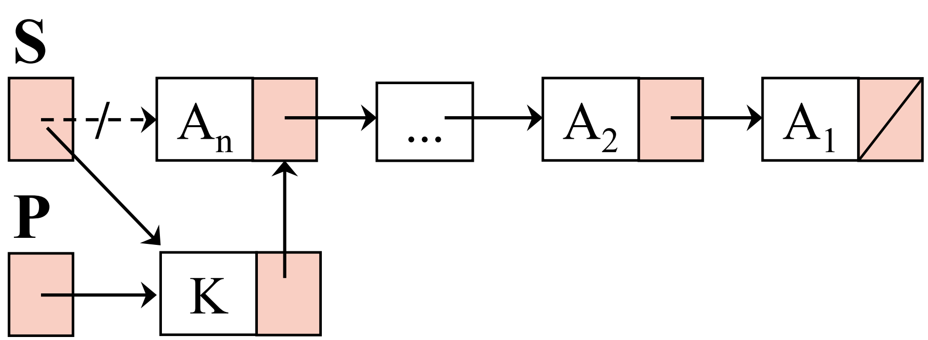



```
bool isEmpty (Stack S){
   return (S==NULL);
}
```

LIFO (Stack) implementation

- **Operations: Push**

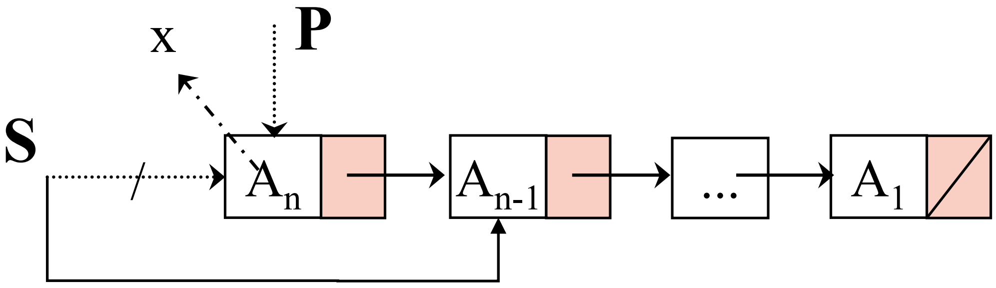
```
void Push (Type K, Stack & S){
   PNode P;
   P = new PNode;
   P->info = K;
   P->next = S;
   S = P;
}
```



LIFO (Stack) implementation

- ## Operations: Pop

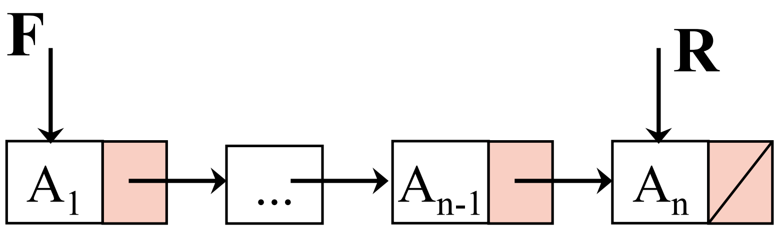
```
PNode Pop (Type & x, Stack & S) {
   PNode P;
   if  (isEmpty (S)) return NULL;
   else {
       P = S;
       x = P->info;
       S = S->next;
       delete P;
       return S;
   }
}
```



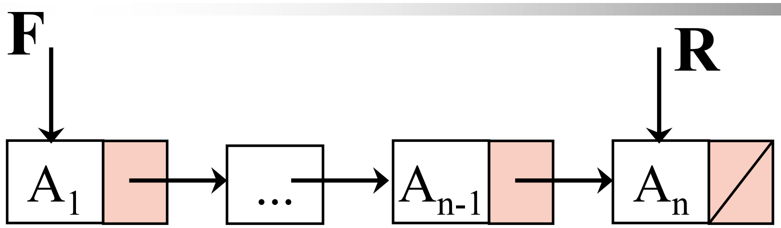
FIFO (Queue) implementation

- **Organization:**
  - Using singly linked list with two access points F (front) and R (rear)
  - Definition:

```
struct  Node {
   Type info;
   Node* next;
};
typedef Node* PNode;
typedef struct {
   PNode F, R;
} Queue;
```



FIFO (Queue) implementation

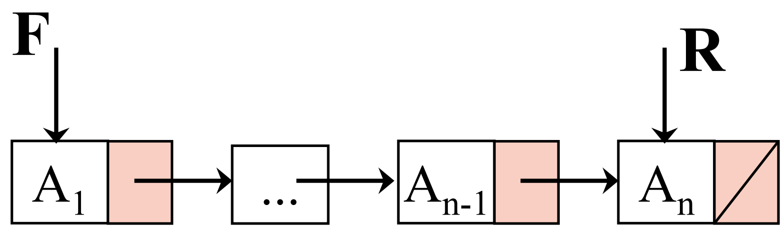


- **Operations:**
  - **Initialize**
  - **IsEmpty**
  - **IsFull**
  - **Enqueue:** add new element into queue
  - **Dequeue:** remove an element from queue

```
void Initialize (Queue & Q){
  Q.F = Q.R = NULL;
}
```

FIFO (Queue) implementation

- **Operations:**



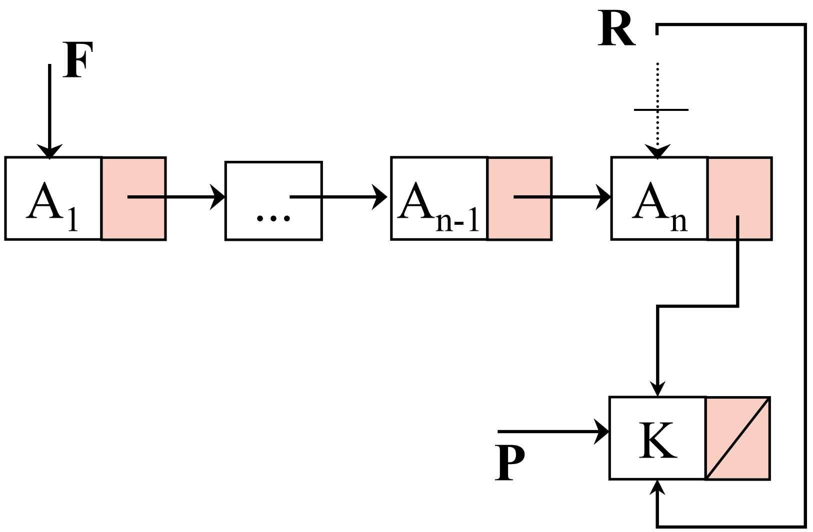

```
bool IsFull (Queue Q) {
  return false;
}
```

```
bool IsEmpty (Queue Q) {
  return  (Q.F == NULL);
}
```

FIFO (Queue) implementation

- **Operations:**

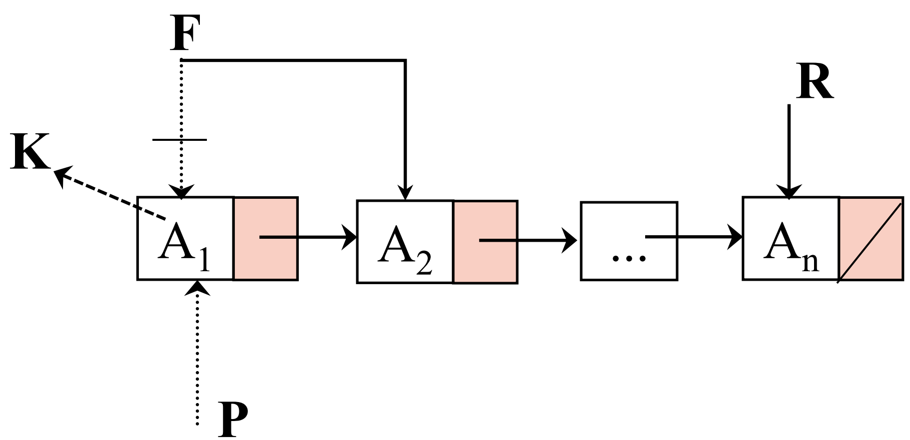
```
void Enqueue(Type K, Queue & Q){
   PNode P;
   P = new PNode;
   P->info = K;
   P->next = NULL;
   if  (isEmpty (Q)) {
        Q.F = Q.R = P;
   }
   else      {
        Q.R->next = P;
        Q.R = P;
   }
}
```



FIFO (Queue) implementation

- **Operations:**

```
void Dequeue(Type & K, Queue & Q){
   Pnode P;
   if  (isEmpty (Q)) return;
   else {
        P = Q.F;
        K = Q.F->info;
        Q.F = Q.F->next;
        delete P;
   }
}
```



Comparison of implementation methods for lists: Sequential Storage vs Linked Storage

- **By memory**
  - For large lists, using linked storage is better than using sequential storage.
- **By complexities of operations:**
  - With linked storage, most operations are more difficult than using sequential storage
  - Using SS are better for searching operations
  - Using LS are better for inserting/removing elements in lists

Exercises

- **Exc 1: Implementation of general list by doubly linked storage. It requires:**
  - Organization of the list
  - Definition of list
  - Implementation of basic operations such as: initialize, insert a new element, remove an element.
- **Exc 2: Definition of two classes Stack and Queue with suitable data and function members using singly linked storage.**
- **Exc 3: Implementation of Queue by double linked storage. It requires:**
  - Organization of the Queue
  - Definition of Queue
  - Implementation of basic operations such as: initialize, enqueue, dequeue

Exercises

- **Exc 4: list of subjects. Each subject consists of following data: subject code, subject name, number of credit. The list is always sorted by the number of credit. You are required to implement the list as follows:**
  - Using singly linked storage
  - The list has basic operations such as: initialize, insert a new subject, remove a subject with given subject code, print the content of list.