TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

2

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 3. Basic data structures

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

What is list ?

- Data structures store objects in a linear structure
- Specify the first object of the list
- Each object: has a unique successor

Contents

1. Array
2. Linked List
3. Stack
4. Queue

5

Contents

1. Array

2. Linked List
3. Stack
4. Queue

6

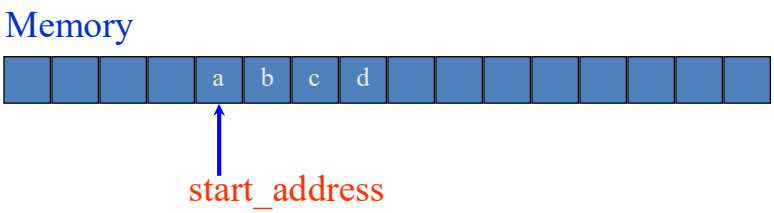
1. Array

- Access elements via indices
- Elements are allocated continuously in memory
- Insertions and removals need to consolidate elements

1. Array

- Access elements via indices
- **Elements are allocated continuously in memory**
- Insertions and removals need to consolidate elements

1D Array Representation in C



Example: 1-dimensional array x = [a, b, c, d]

- Allocated at contiguous memory locations
- Location($x[i]$) = start_address + W*i

where

- start_address: the address of the first element in the array
- W: size of each element in the array

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example

Write a C program that gives the address of each element of an 1D array:

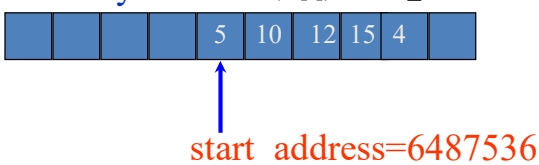
```
#include <stdio.h>
int main()
{   int A[ ] = {5, 10, 12, 15, 4};
     int rows=5;
     /* print the address of 1D array by using pointer  */
   printf("Address    Contents\n");
   for (int i=0; i < rows; i++)
        printf("%8u %5d\n", &A[i], A[i]);
}
```

Result in DevC
(sizeof(int)=4)

```
Address    Contents
6487536       5
6487540      10
6487544      12
6487548      15
6487552       4
```

ptr+i : address of element A[i]
*(ptr+i) : content of element A[i]

Memory Location(A[i]) = start_address + W*i



Result in turboC
(sizeof(int)=2)

```
Address    Contents
  65516       5
  65518      10
  65520      12
  65522      15
  65524       4
```

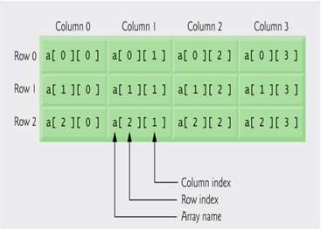
Two-dimensional array

- How to declare:

```
<element-type> <arrayName> [size1][size2];
```

Example: `double a[3][4];`

may be shown as a table



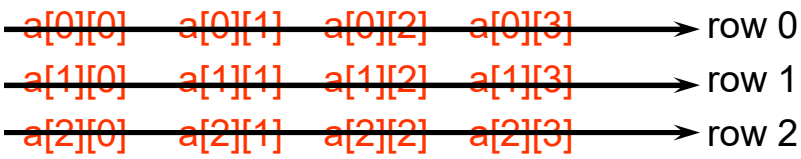
- Using the two-dimensional array initializer

Example: `int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

- Access to element of array: a[2] [1];

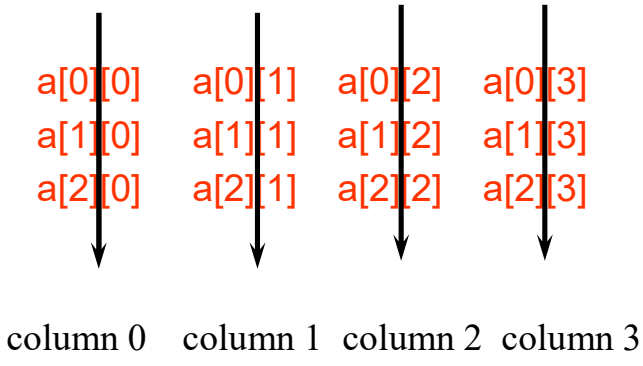
a[0][0] = 1	a[0][1]=2	a[0][2]=3	a[0][3]=4
a[1][0] = 5	a[1][1]=6	a[1][2]=7	a[1][3]=8
a[2][0] = 9	a[2][1]=10	a[2][2]=11	a[2][3]=12

Rows of a 2D Array



NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Columns of a 2D Array



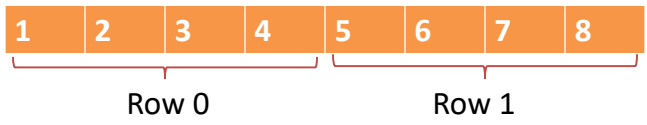
NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Multidimensional Arrays

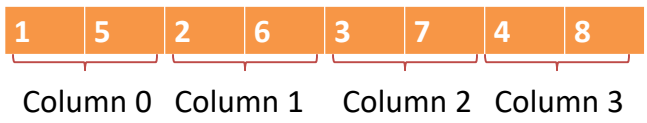
- Multidimensional arrays are usually implemented by one dimensional array via either row major order or column major order.

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8

User's view



Row-major storage

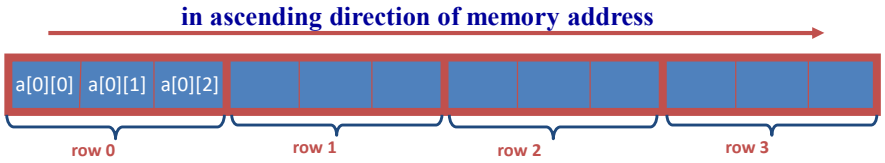


Column-major storage

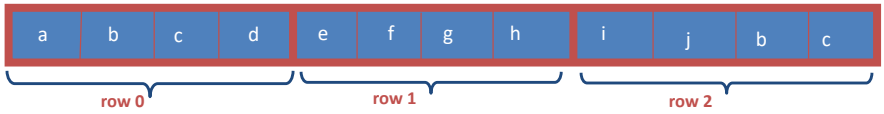
Row-Major Order (e.g. C/C++, Python by default, Pascal)

- Row- major order is a method of representing multi-dimensional array in sequential memory. In this method, elements of an array are arranged sequentially row by row. Thus, elements of the first row occupies the first set of memory locations reserved for the array, elements of the second row occupies the next set of memory and so on.

- Example: int a[4][3]



- Example 3 x 4 array: a b c d
  e f g h
  i j k l

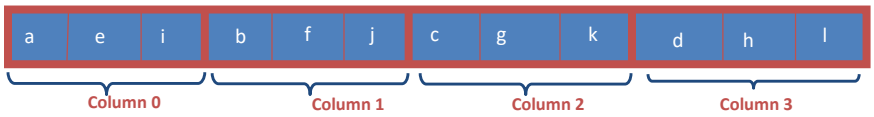


Column-Major Order (e.g. Matlab, Fortran)

- In this method, elements of an array are arranged sequentially column by column. Thus, elements of the first column occupies the first set of memory locations reserved for the array, elements of the second column occupies the next set of memory and so on.

- Example 3 x 4 array:

a b c d
e f g h
i j k l



Locating Element x[i][j]: row-major order

- Assume x:
  - has ***r*** rows and ***c*** columns (thus, each row has ***c*** elements)

- Locating element $x[i][j]$:
  - i rows to the left of row i ➔ so $i*c$ elements to the left of $x[i][0]$
  - $x[i][j]$ is mapped to position: $i*c + j$ of the 1D array
  - The location of element $x[i][j]$:

  Location($x[i][j]$) =start_address + W ($i*c + j$)

  Where
  - start_address: the address of the first element ($x[0][0]$) in the array
  - W: is the size of each element
  - c: number of columns in the array
  - r: number of rows in the array

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example

Write a C program that gives the address of each element of a 2D array:

```
#include <stdio.h>
int main()
{  int  a[3] [4] = {1,2,3,4,5,6,7,8,9,10,11,12};
   int rows=3, cols =4;
   /* print the address of 2D array by using pointer  */
   int *ptr = a;
   printf("Address    Contents\n");
   for (int i=0; i < rows; i++)
      for (int j=0; j < cols; j++)
       printf("%8u %5d\n", &a[i][j], a[i][j]);
}
```

Result in DevC (sizeof(int)=4)

Address	Contents
6487488	1
6487492	2
6487496	3
6487500	4
6487504	5
6487508	6
6487512	7
6487516	8
6487520	9
6487524	10
6487528	11
6487532	12

Memory

Location(a[i][j]) = start_address + W*[(i*cols) + j]

start_address=6487488

Location(a[1][2]) = ?

Locating Element x[i][j]: column-major order

- Assume x:

.....	r Elements of column 0	r Elements of column 1	r Elements of column 2	....	r Elements of columns i	.....

  - has ***r*** rows and ***c*** columns (thus, each column has ***r*** elements)
- Locating element $x[i][j]$:
  - j columns to the left of column j ➔ so $j*r$ elements to the left of $x[0][j]$
  - $x[i][j]$ is mapped to position: $j*r + i$ of the 1D array
  - The location of element $x[i][j]$:

    Location($x[i][j]$) = start_address + W ($j*r + i$)

    Where
    - start_address: the address of the first element in the array
    - W: is the size of each element
    - c: number of columns in the array
    - r: number of rows in the array

Example: array : `int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};`

Determine the address of the element a[1][2] if start_address = 6487488

Example 1: Row- and Column-Major Orders

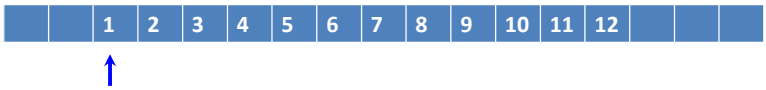
2D array:

int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};

a[0][0] = 1	a[0][1]=2	a[0][2]=3	a[0][3]=4
a[1][0] = 5	a[1][1]=6	a[1][2]=7	a[1][3]=8
a[2][0] = 9	a[2][1]=10	a[2][2]=11	a[2][3]=12

Memory: row-major order

Location(a[i][j]) = start_address + W*[(i*cols) + j]



start_address=6487488

Location(a[1][2]) = ?

Memory: column-major order

Location(a[i][j]) = start_address + W*[(j*rows) + i]

start_address=6487488

Location(a[1][2]) = ?

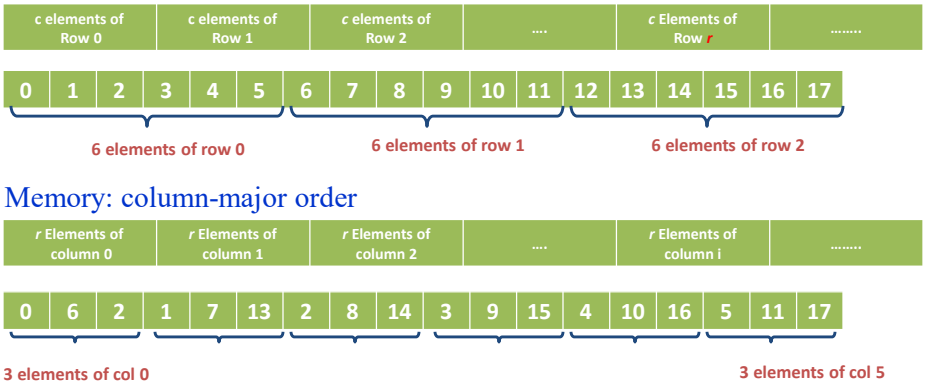
Example 2: Row- and Column-Major Orders

2D array: *r* rows, *c* columns

Example: int a[3][6]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};

a[0][0]=0 a[0][1]=1 a[0][2]=2 a[0][3]=3 a[0][4]=4 a[0][5]=5
a[1][0]=6 a[1][1]=7 a[1][2]=8 a[1][3]=9 a[1][4]=10 a[1][5]=11
a[2][0]=12 a[2][1]=13 a[2][2]=14 a[2][3]=15 a[2][4]=16 a[2][5]=17

Memory: row-major order start_address = 1000 ➔ Location(a[1][4]) = ?



Memory: column-major order

Example

We have stored the two-dimensional array students in memory. The array is 100 × 4 (100 rows and 4 columns). Show the address of the element students[5][3] assuming that the element student[0][0] is stored in the memory location with address 1000 and each element occupies only two bytes memory location. The computer uses row-major storage.

Solution:

1. Array

- Access elements via indices
- Elements are allocated continuously in memory
- **Insertions and removals need to consolidate elements**

23

Operations on the array

- The common operations on arrays are searching, insertion, deletion, retrieval and traversal.

Example: Given an array *S* consists of *n* integers: S[0], S[1], …, S[*n*-1]

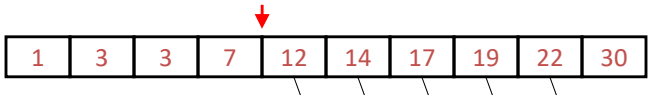
- **Search operation**: search a value ***key*** whether appears in the array S or not
  `function Search(S, key)` returns true if ***key*** appears in S; false otherwise
- **Retrieval operation**: get the value of the element at index ***i*** of the array S
  `function Retrieve(S, i)`: returns the value S[i] if 0 <= *i* <= *n*-1
- **Traversal operation**: print the value of all elements in the array S
  `function PrintArray(S, n)`
- **Insert operation**: insert a value ***key*** into the array S
- **Delete operation**: delete the element at index ***i*** of the array S

- Although searching, retrieval and traversal of an array is an easy job, insertion and deletion is time consuming. The elements need to be shifted down before insertion and shifted up after deletion.

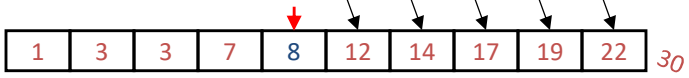
NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Inserting an element into an array

- Assume we need to insert 8 into an array already be sorted in ascending order:



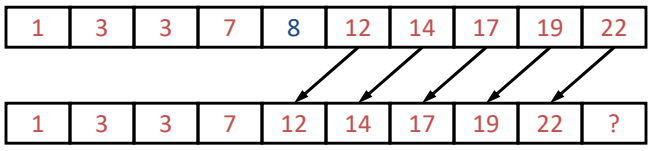
- We can do it by shifting to the right one cell for all the elements after the mark
  - It thus need to remove 30 from the array



• Moving all elements of the array is a slow operation (requires linear time O(*n*) where *n* is the size of array)

Deleting an element from an array

- In order to delete an element, we need to shift to the left all previous elements



- Delete operation is a slow operation.
- Regular implementation of this operation is undesirable.
- Delete operation makes the last element free
  - How we could mark the last element of the array being free?
    - We need variable to store the size of the array

    Example: variable size is used to store the size of the array. Before deletion, size = 10. After deletion, we need to update the value of size: size = 10 – 1 = 9

Operations on the array

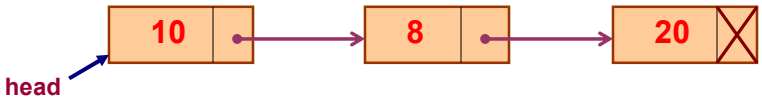
Thinking about the operations discussed in the previous section gives a clue to the application of arrays. If we have a list in which a lot of insertions and deletions are expected after the original list has been created, we should not use an array. An array is more suitable when the number of deletions and insertions is small, but a lot of searching and retrieval activities are expected.

An array is a suitable structure when a small number of insertions and deletions are required, but a lot of searching and retrieval is needed.

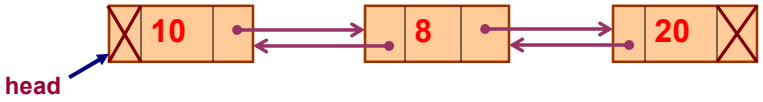
Contents

2. Linked list

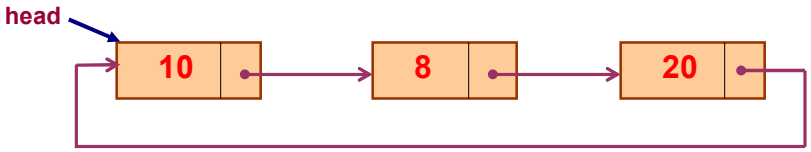
- Singly linked list



- Doubly linked list

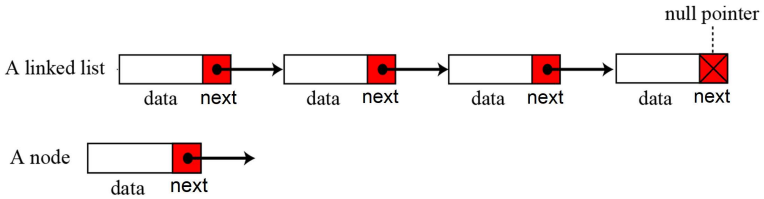


- Circular linked list

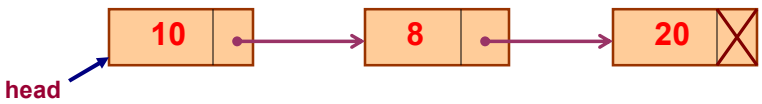


Singly Linked list

- A singly linked list is a sequences of nodes, each node contains 2 parts: data and reference (address) to the next node.
- Example: Figure shows a singly linked list contains four nodes:



- Keeping track of a singly linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.)
  - If head is NULL, the singly linked list is empty



NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Singly Linked list

- Before further discussion of singly linked lists, we need to explain the notation we use in the figures. We show the connection between two nodes using a line. One end of the line has an arrowhead, the other end has a solid circle.

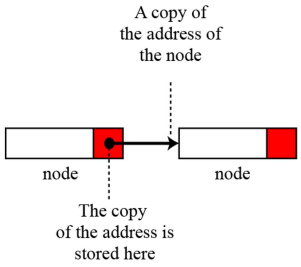
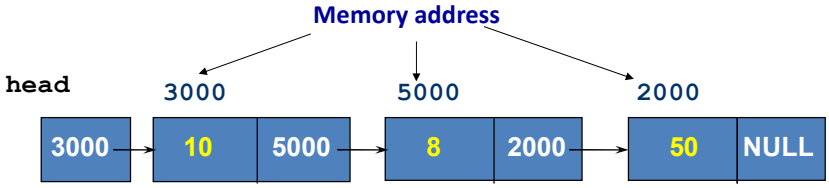
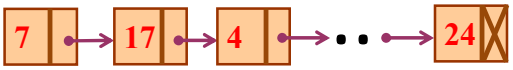


Figure. The concept of copying and storing pointers

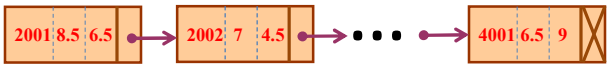


Declare singly linked list in C programming language

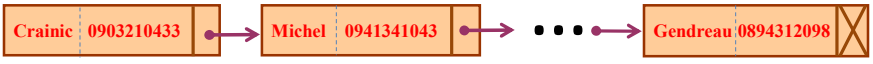
- List of integer numbers:



- List of students with data: student's ID, grade of math and physics



- List of contacts with data: name, phone number



➔ Need to declare:

- the type of data in the node first,
- then the singly linked list consists of (1) data of the node, and (2) the pointer to store the address of the next node in the list

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Declare singly linked list

Need to declare:
- the type of data in the node first,
- then the linked list consists of (1) data of the node, and (2) the pointer to store the address of the next node in the list

```
typedef struct {
   .....
}NodeType;

typedef struct {
   NodeType data;
   struct node* next;
}node;
node* head;
```

Define the type of data of the node

Define the singly linked list

This declaration define `node` which is a record consisting of 2 fields:
- `data` : stores data of node, has the type `NodeType` (which was defined in `typedef…NodeType`, and could consist of several attributes)
- `next` : the pointer which stores the address of the next node in the list

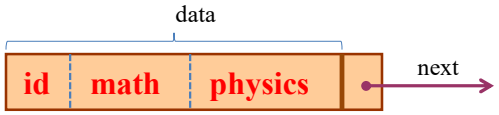
Pointer `head` : store address of the first node in the list

Example1: List of students with data: id of student, marks of 2 subjects: math, physics

```
typedef struct{
    char id[15];
    float math, physics;
}student;
```

Define the type of data of the node

```
typedef struct {
   student data;
   struct node* next;
}node;
node* head;
```



A node

Declare singly linked list

```
typedef struct {
   .....
}NodeType;

typedef struct {
   NodeType data;
   struct node* next;
}node;
node* head;
```

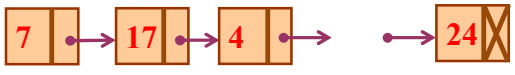
Define the type of data of the node

Define the singly linked list

This declaration define `node` which is a record consisting of 2 fields:
- `data` : stores data of node, has the type `NodeType` (which was defined in `typedef…NodeType`, and could consist of several attributes)
- `next` : the pointer which stores the address of the next node in the list

Pointer `head` : store address of the first node in the list

Example2: List of integer numbers




```
typedef struct {
    int data;
   struct node* next;
}node;
node* head;
```

"`int`" is the type of node, so do not need to use "`typedef….NodeType`" to define the type

A node

Declare singly linked list

Need to declare:
- the type of data in the node first,
- then the linked list consists of (1) data of the node, and (2) the pointer to store the address of the next node in the list

```
typedef struct {
   .....
}NodeType;

typedef struct {
   NodeType data;
   struct node* next;
}node;
node* head;
```

Define the type of data of the node

Define the singly linked list

This declaration define `node` which is a record consisting of 2 fields:
- `data` : stores data of node, has the type `NodeType` (which was defined in `typedef…NodeType`, and could consist of several attributes)
- `next` : the pointer which stores the address of the next node in the list

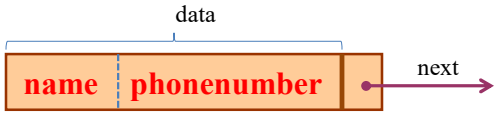
Pointer `head` : store address of the first node in the list

Example 3: List of contacts with data: name and phone number

```
typedef struct{
    char name[15];
    char phone[20];
}contact;
```

```
typedef struct {
   contact data;
   struct node* next;
}node;
node* head;
```

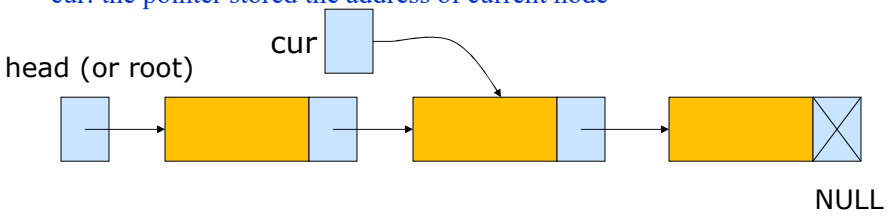
Define the type of data of the node



A node

Important elements of singly linked list

- head: store the address of the first node in the linked list
- NULL: value of the pointer of the last node in the linked list
- cur: the pointer stored the address of current node



- Allocate memory for a new node pointed by the pointer `newNode` in the list:

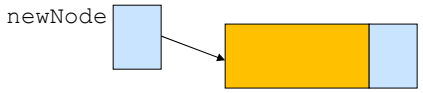
```
node *newNode = (node *) malloc(sizeof(node));
```

- Access to the data of the node pointed by pointer `newNode` :

```
newNode->data          (*newNode).data
```

- Free memory allocated for node pointed by pointer `newNode` :

```
free(newNode);
```



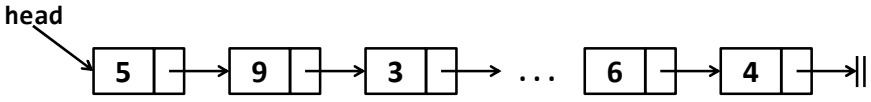
36

Operations on singly linked Lists

- Traverse the singly linked list
- Insert a node into the singly linked list
- Delete a node from the singly linked list
- Search data in the singly linked list

Assumption

```
typedef struct {
    int data;
    struct Node* next; // pointer to the successor
} Node;
Node* head; // pointer to the first element of the list
```



Create a new node

```
#include <stdio.h>
typedef struct Node{
    int data;
    struct Node* next; // pointer to the successor
}Node;

Node *makeNode(int v)  //allocate memory for a new node
{
    Node* p = (Node*)malloc(sizeof(Node));
    //Node *p = new Node;
    p->data = v; p->next = NULL;
    return p;
}
```

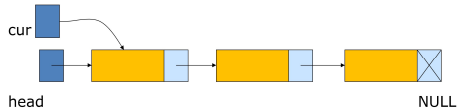

Operations on singly linked Lists

- **Traverse the singly linked list**
- Insert a node into the singly linked list
- Delete a node from the singly linked list
- Search data in the singly linked list

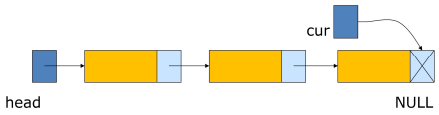
Traversing a singly linked list: 2nd way

```
void printList(Node* head)
{
  Node* cur = head;
  for (cur = head; cur != NULL; cur = cur->next)   printf("%d ",cur->data);
}
```

```
void printList(Node* head)
{
  Node* cur = head;
  while(cur != NULL){
    printf("%d ",cur->data);
    cur = cur->next;
  }
}
```



- Change the value of the pointer cur
- Finish to browse the list when the NULL value is encountered



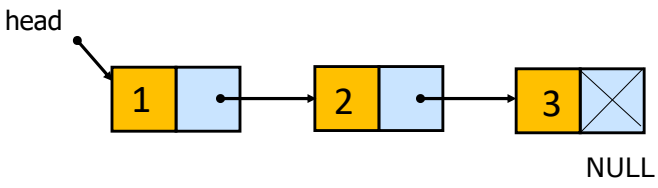
Exercise 1

- A sequence of integers is stored by a singly linked list.

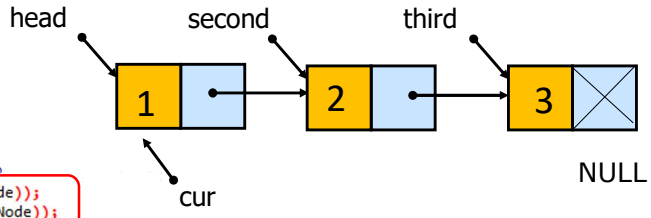
```
typedef struct Node{
    int data;
    struct Node *next;
}Node;
Node *head;
```

a. Create a list stored 3 integers: 1, 2, 3

b. Print the list of these 3 integers



```c
#include<stdio.h>

typedef struct
{
  int data;
  struct Node *next;
}Node;

void showData_Of_Current_Element(int data){
   printf("Data = %d\n",data);
}

int main()
{
  struct Node* head;
  Node* second = NULL;
  Node* third = NULL;

  //allocate memory for 3 nodes in the heap
  head = (Node*)malloc(sizeof(Node));
  second = (Node*)malloc(sizeof(Node));
  third = (Node*)malloc(sizeof(Node));

  head->data = 1; //assign data for the first node in the list
  head->next = second; //connect the first node to the second node

  second->data = 2; //assign data for the 2nd node in the list
  second->next = third; //connect the 2nd node to the 3rd node

  third->data = 3; //assign data for the 3rd node
  third->next = NULL;

  Node* cur;
  for (cur = head; cur != NULL; cur = cur->next)
     showData_Of_Current_Element(cur->data);
  return 0;
}
```



Data = 1

Data = 2

Data = 3

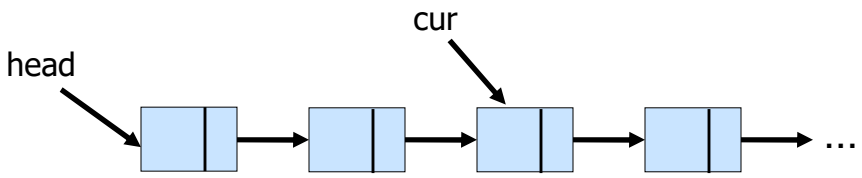
Operations on singly linked lists

- Traverse the singly linked list
- **Insert a node into the singly linked list**
- Delete a node from the singly linked list
- Search data in the singly linked list

Operations on singly linked list: Insertion

Insert a new node :

- At the beginning of the list
- After the position pointed by the pointer cur
- Before the position pointed by the pointer cur
- At the end of the list

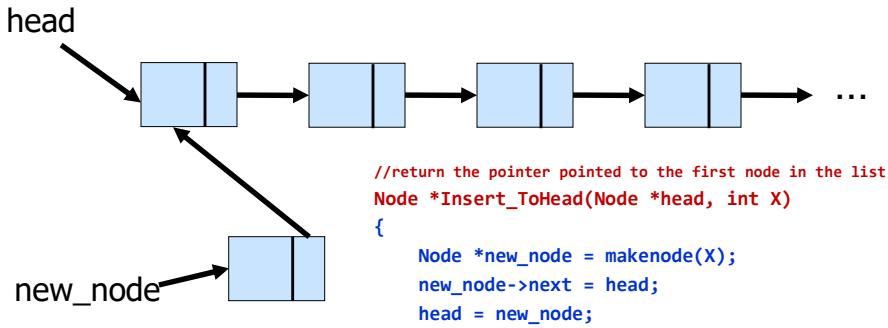


Operations on singly linked list: Insertion

Insert a new node:

- At the beginning of the list

```
<create a new node new_node>;
new_node ->next = head;
head= new_node;
```




```
//return the pointer pointed to the first node in the list:
Node *Insert_ToHead(Node *head, int X)
{
    Node *new_node = makenode(X);
    new_node->next = head;
    head = new_node;
    return head;
}
```

Operations on singly linked list: Insertion

```
//return the pointer pointed to the first node in the list:
Node *Insert_ToHead(Node *head, int X)
{
    Node *new_node = makenode(X);
    new_node->next = head;
    head = new_node;
    return head;
}
```

```
void Insert_ToHead(Node **head_ref, int X)
{
    Node *new_node = makenode(X);
    new_node->next = *head;
    *head = new_node;
}
```


Exercise

Ask the user to enter an integer *n*, then enter *n* integers numbers from the keyboard. Each time an integer is read, insert it at the beginning of a linked list. Finally, prints on the screen all values stored in that linked list.

```
typedef struct Node{
    int data;
    struct Node *next;
}Node;
Node *head;
```

```
struct Node {
    int data;
    struct Node *next;
};
struct Node *head;
```

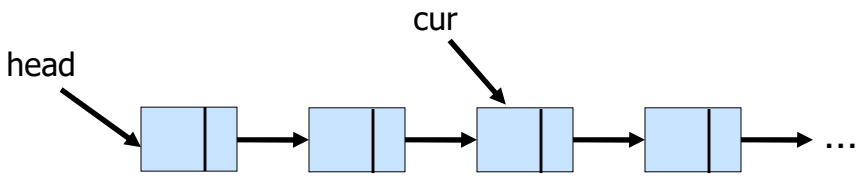
```
void Insert_ToHead(Node** head_ref, int new_data);
node *Insert_ToHead(Node *head, int new_data); /*return the pointer pointed to the first node in the list*/
```

```
Enter n = 4
Enter the number 1: 1
Enter the number 2: 3
Enter the number 3: 5
Enter the number 4: 7
Values in the linked list:  7  5  3  1
```

Operations on singly linked list: Insertion

Insert a new node :

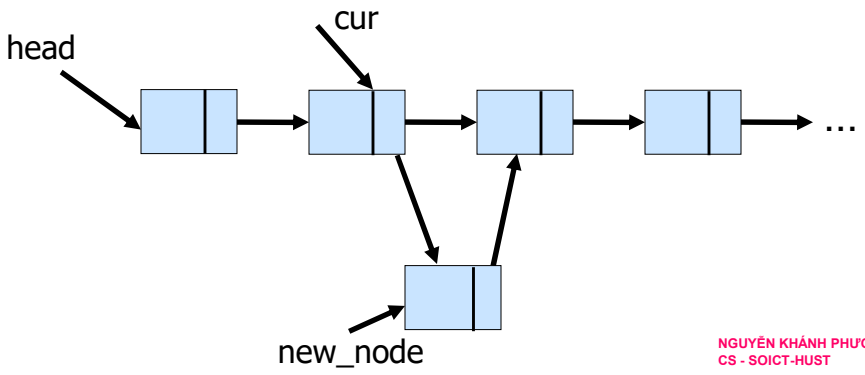
- At the beginning of the list
- **After the position pointed by the pointer cur**
- Before the position pointed by the pointer cur
- At the end of the list



Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```



Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```

Write a function to insert a node with data = *X* (having the type **int** after the node pointed by the pointer **cur**. The function returns the address of the new node:

```
Node *Insert_After(Node *cur, int X)
{
   Node *new_node = makenode(X);   //(1)
   new_node->next = cur->next;     //(2)
   cur->next = new_node;           //(3)
   return new_node;
}
```

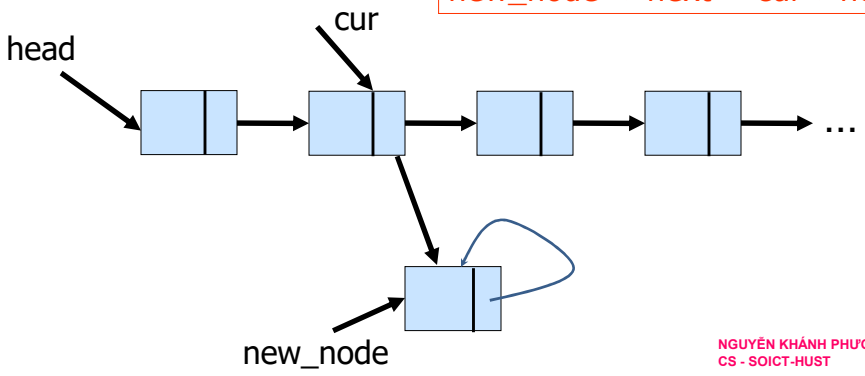

Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```

?? Empty list

```
// wrong implementation:
cur->next = new_node;
new_node ->next = cur->next;
```



Operations on singly linked list: Insertion

- Insert a new node after the node pointed by the pointer cur:

```
<create a new node new_node>;
new_node ->next = cur->next;
cur->next = new_node;
```

?? Empty list

```
<create a new node new_node>;
if (head == NULL) { /* list does not have any node yet */
    head = new_node;
}
else {
    new_node ->next = cur->next;
    cur->next = new_node;
}
```

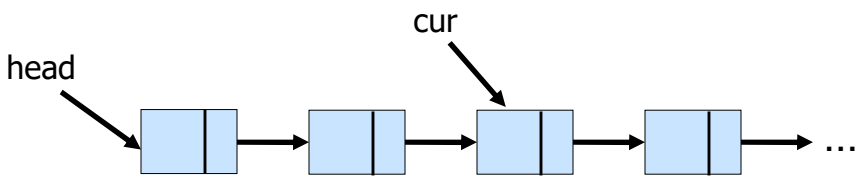
```
Node *Insert_After(Node *cur, int X)
{
   Node *new_node = makenode(X);  //(1)
   if (head == NULL) head = new_node;
   else {
      new_node->next = cur->next;    //(2)
      cur->next = new_node;          //(3)
    }
    return new_node;
}
```

53

Operations on singly linked list: Insertion

Insert a new node :

- At the beginning of the list
- After the position pointed by the pointer cur
- **Before the position pointed by the pointer cur**
- At the end of the list



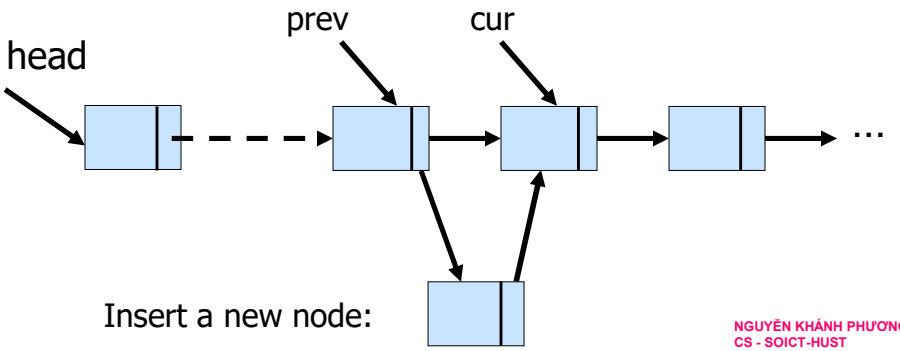
Operations on singly linked list: Insertion

Insert a new node **before the node pointed by the pointer cur**

```
<Determine pointer prev pointed to
the predecessor of cur>;
<create a new node new_node>;
prev->next = new_node;
new_node->next = cur;
```

?? List does not have any node yet

?? cur is the first node in the list



Operations on singly linked list: Insertion

Insert a new node **before the node pointed by the pointer cur**

```
<Determine pointer prev pointed
to the predecessor of cur>;
<create a new node new_node>;
prev->next = new_node;
new_node->next = cur;
```

?? List does not have any node yet

?? cur is the first node in the list

```
<Determine pointer prev pointed to the predecessor of cur>;
<create a new node new_node>;
if (head == NULL) { /* list does not have any node yet */
    head = new_node;
}
else if (cur == head) { //cur is the first node in the list
    head = new_node;
    new_node->next = cur;
}
else {
    prev->next = new_node;
    new_node->next = cur;
}
```

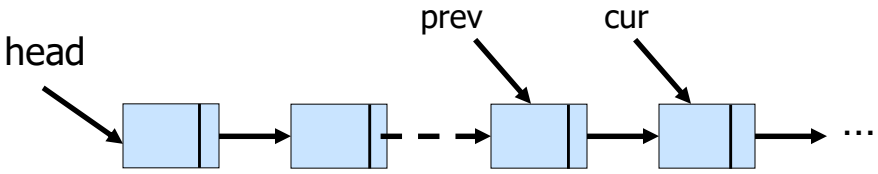
`Node *Insert_Before(Node *head, Node *cur, int X)`

Operations on singly linked list: Insertion

Insert a new node **before the node pointed by the pointer cur**

```
<Determine pointer prev pointed to the predecessor of cur>;
<create a new node new_node>;
prev->next = new_node;
new_node->next = cur;
```

```
node *prev = head;
while (prev->next != cur) prev = prev->next;
```



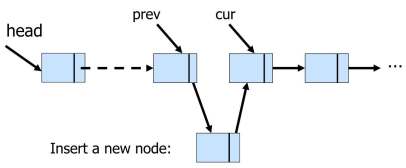
Node *Insert_Before(Node *head, Node *cur, int X)

57

Operations on singly linked list: Insertion

Insert a new node **before the node pointed by the pointer cur**

```
Node *Insert_Before(Node *head, Node *cur, int X)
{
    Node *prev = head;
    while (prev->next != cur) prev = prev->next;
    Node *new_node = makenode(X);
    if (head == NULL)  /* list does not have any node yet */
       head = new_node;
    else if (cur == head) { //cur is the first node in the list
      head = new_node;
      new_node->next = cur;
    }
    else {
      prev->next = new_node;
      new_node->next = cur;
    }
    return new_node;
}
```

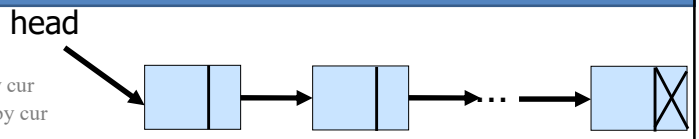


NGUYỄN KHÁNH PHƯƠNG
KHMT – SOICT - ĐHBK HN

Operations on singly linked list: Insertion

Insert a new node:

- At the beginning
- After the node pointed by cur
- Before the node pointed by cur
- **At the end of the list**




```
<create a new node new_node>;
if (head == NULL) { /* list does not have any node yet*/
     head = new_node;
}
else {
     //move the pointer to the end of the list:
      node *last =head;
      while (last->next != NULL) last = last->next;
      //Change the pointer next of the last node:
    last->next = new_node;
}
```

Complexity is ….

```
Node *Insert_ToLast(Node *head, int X)
{
   Node *new_node = makenode(X);
   if (head == NULL) head = new_node;
   else  {
      Node *last = head;
      while (last->next != NULL) last = last->next; //move to the last node
      last->next = new_node;
   }
   return head;
}
```

59

Operations on singly linked list: Insertion

```
Node *Insert_ToLast(Node *head, int X)
{
   Node *new_node = makenode(X);
   if (head == NULL) head = new_node;
   else  {
      Node *last = head;
      while (last->next != NULL) last = last->next; //move to the last node
      last->next = new_node;
   }
   return head;
}
```

```
void Insert_ToLast(Node **head_ref, int X)
{
   Node *new_node = makenode(X);
   if (*head == NULL) *head = new_node;
   else {
      Node *last = *head;
      while (last->next != NULL) last = last->next;  //move to the last node
      last->next = new_node;
   }
}
```

NGUYỄN KHÁNH PHƯƠNG
KHMT – SOICT - ĐHBK HN

Operations on singly linked Lists

- Traverse the singly linked list
- Insert a node into the singly linked list
- **Delete a node from the singly linked list**
- Search data in the singly linked list

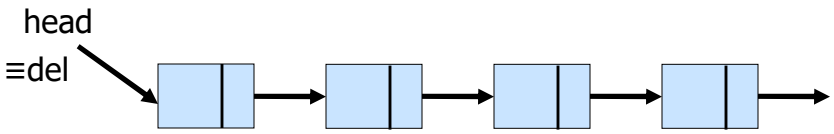
Operations on singly linked lists: Deletion

- **Delete a node**
- Delete all nodes of the list

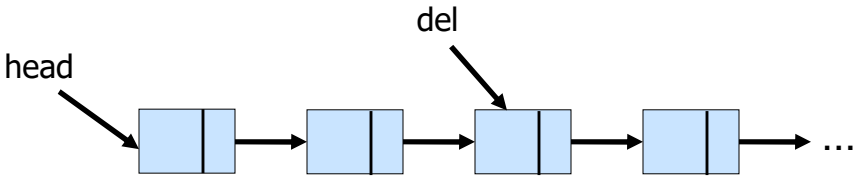
Operations on singly linked lists: Deletion

Delete a node:

- The first node of the list



- Middle/last node of the list

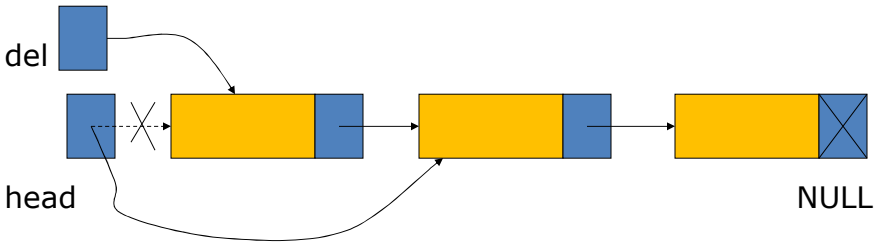


63

Delete the first node of the list

- Delete the node `del` that is currently the first node of the list:

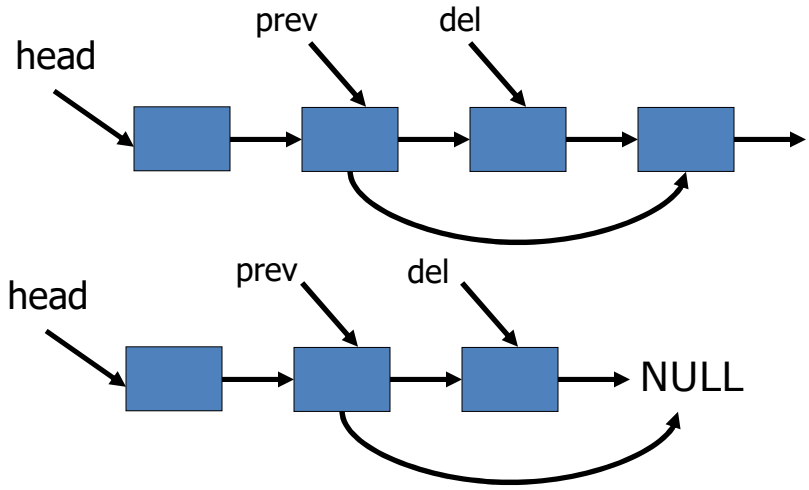
```
head = del->next;
free(del);
```



Delete the node at the middle/end of the list

Delete node `del` that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory
```



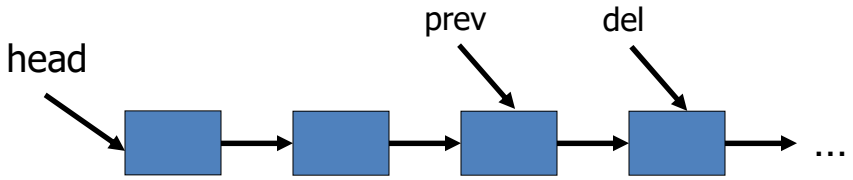
65

Delete the node at the middle/end of the list

Delete node `del` that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory)
```

```
Node *prev =head;
while (prev->next != del) prev = prev->next;
```



66

Delete a node pointed by the pointer del

Write the function **node *Delete_Node(node *head, node *del)**
to delete a node pointed by the pointer "del" of the list with the first node pointed by the pointer "head".
The function returns the address of the first node in the list after deletion:

- Delete the node del that is currently the first node of the list:

```
head = del->next;
free(del);
```

Delete node del that is currently the middle/last node of the list:

```
<Determine the pointer prev pointed to the previous node of del>;
prev->next = del->next; //modify the link
free(del); //delete node del to free memory)
```

```
Node *prev =head;
while (prev->next != del) prev = prev->next;
```

```
Node *Delete_Node(Node *head, Node *del)
{
   if (head == del) //del is pointed to the first node of the list:
   {
      head = del->next;
      free(del);
   }
   else{//del is pointed to the middle/last node of the list
      Node *prev = head;
      while (prev->next != NULL) prev = prev->next;
      prev->next =del->next;
      free(del);
   }
   return head;
}
```

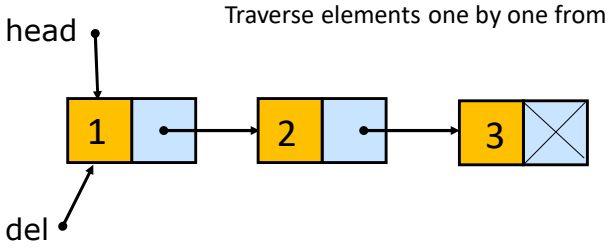

Operations on singly linked lists: Deletion

- Delete a node
- **Delete all nodes of the list**

Freeing all nodes of a list

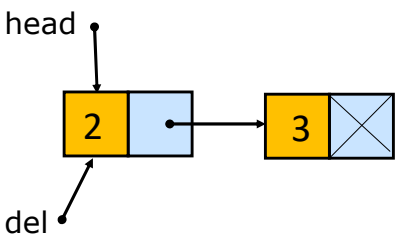
```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```

Traverse elements one by one from the head till the end



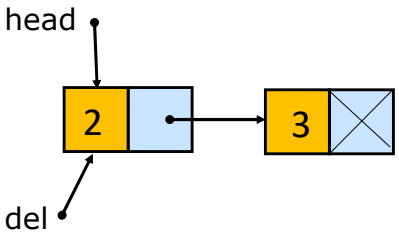
Freeing all nodes of a list

```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```



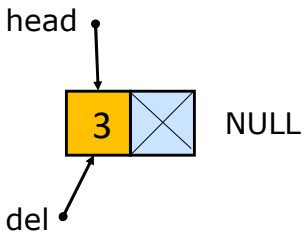
Freeing all nodes of a list

```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```



Freeing all nodes of a list

```
del = head ;
while (del != NULL)
{
    head = head->next;
    free(del);
    del = head;
}
```



Freeing all nodes of a list

Write the function `Node* deleteList(Node* head)`
to delete all the node in the list having the first node pointed by the pointer `head`
The function returns the pointer `head` after deletion

```
Node* deleteList(Node* head)
{
   Node *del = head ;
   while (del != NULL)
   {
      head = head->next;
      free(del);
      del = head;
   }
   return head;
}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Check whether the singly linked list is empty or not

Write the function `int IsEmpty(Node *head)`
to check whether the singly linked list is empty or not (the pointer head pointed to the first node of the list).
The function returns 1 if the list is empty; 0 otherwise

```
int IsEmpty(Node *head) {
     if (head == NULL)
          return 1;
     else return 0;
}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Operations on singly linked Lists

- Traverse the singly linked list
- Insert a node into the singly linked list
- Delete a node from the singly linked list
- **Search data in the singly linked list**

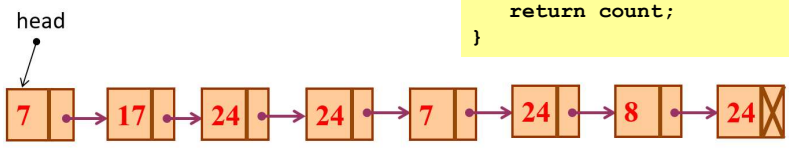
Searching

- To search for an element, we traverse from head until we locate the object or we reach the end of the list.

Example: Given a linked list consisting of integer numbers. Count the number of nodes with data field equal to number x.

```
typedef struct {
    int data;
    struct node* next;
}Node;
Node* head;
```

```
int countNodes(int x){
    int count = 0;
    Node* e = head;
    while(e != NULL){
        if(e->data == x) count++;
        e = e->next;
    }
    return count;
}
```




```
int Result1 = countNodes(24);

int a =7;
int Result2 = countNodes(a);
```

Result1 = ?
Result2 = ?

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Time Complexity: Singly-linked lists vs. 1D-arrays

Operation	ID-Array Complexity	Singly-linked list Complexity
Insert at beginning	O(n)	O(1)
Insert at end	O(1)	O(1) if the list has **tail** reference O(n) if the list has no **tail** reference
Insert at middle*	O(n)	O(n)
Delete at beginning	O(n)	O(1)
Delete at end	O(1)	O(n)
Delete at middle*	O(n): O(1) access followed by O(n) shift	O(n): O(n) search, followed by O(1) delete
Search	O(n) linear search O(log n) Binary search	O(n)
Indexing: What is the element at a given position k?	O(1)	O(n)

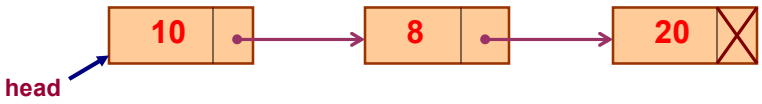
* middle: neither at the beginning nor at the end

Singly-linked lists vs. 1D-arrays

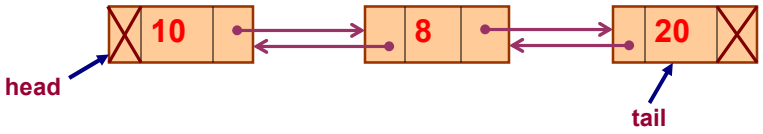
ID-array	Singly-linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Extra storage needed for references; however uses exactly as much memory as it needs
Sequential access is faster because of greater locality of references [Reason: Elements in contiguous memory locations]	Sequential access is slow because of low locality of references [Reason: Elements not in contiguous memory locations]

2.3. Linked list

- Singly linked list

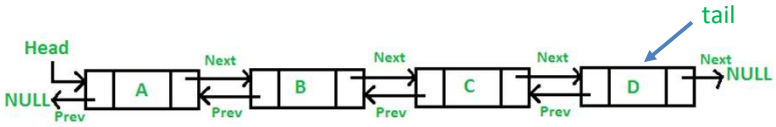


- **Doubly linked list**



Doubly linked list

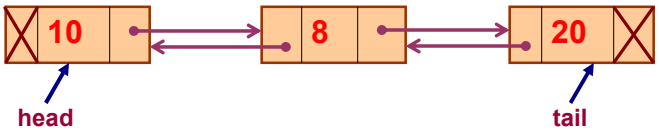
- A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list



- 2 special nodes: `tail` and `head`
  - `head` has pointer `prev = null`
  - `tail` has pointer `next = null`
- Basic operations are considered similar as in the singly linked list

Doubly linked list

- Declare doubly linked list to store integer numbers:

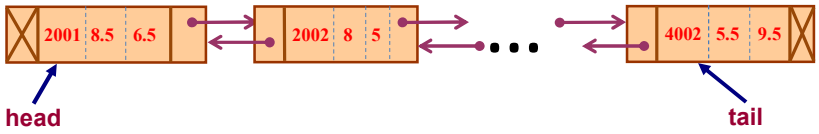



```
typdedef struct dllist {
   int number;
   struct dllist *next;
   struct dllist *prev;
} dllist;
dllist *head, *tail;
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Doubly linked list

- Declare doubly linked list store data of students: ID, marks of math and physics

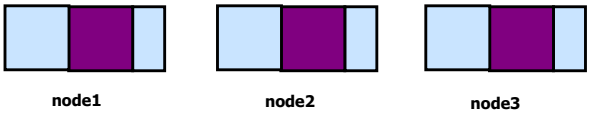



```
typedef struct{
    char id[15];
   float math, physics;
}student;

typedef struct dllist{
   student data;
   struct ddlist* next;
   struct ddlist* prev;
}dllist;
dllist *head, *tail;
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Doubly linked list – Example (C proramming)

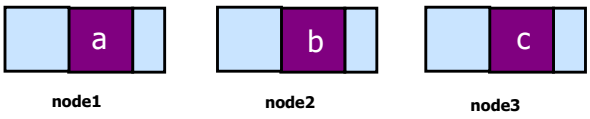
```
typedef struct dllist {
      char data;
      struct dllist *prev;
      struct dllist *next;
}dllist;
dllist *node1, *node2, *node3;
node1 = (dllist*)malloc(sizeof(dllist));
node2 = (dllist*)malloc(sizeof(dllist));
node3 = (dllist*)malloc(sizeof(dllist));

node1->data='a';
node2->data='b';
node3->data='c';
node1->prev=NULL;
node1->next=node2;
node2->prev=node1;
node2->next=node3;
node3->prev=node2;
node3->next=NULL;
dllist *temp;
for (temp = node1; temp != NULL; temp = temp->next)
   printf("%d ",temp->data);
```



83

Doubly linked list – Example (C proramming)

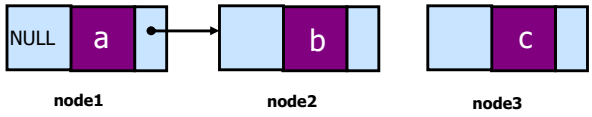
```
typedef struct dllist {
      char data;
      struct dllist *prev;
      struct dllist *next;
}dllist;
dllist *node1, *node2, *node3;
node1 = (dllist*)malloc(sizeof(dllist));
node2 = (dllist*)malloc(sizeof(dllist));
node3 = (dllist*)malloc(sizeof(dllist));

node1->data='a';
node2->data='b';
node3->data='c';
node1->prev=NULL;
node1->next=node2;
node2->prev=node1;
node2->next=node3;
node3->prev=node2;
node3->next=NULL;
dllist *temp;
for (temp = node1; temp != NULL; temp = temp->next)
   printf("%d ",temp->data);
```



84

Doubly linked list – Example (C proramming)

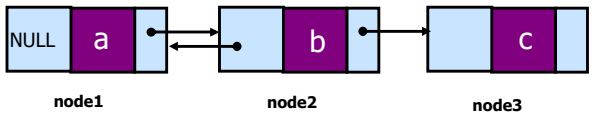
```
typedef struct dllist {
      char data;
      struct dllist *prev;
      struct dllist *next;
}dllist;
dllist *node1, *node2, *node3;
node1 = (dllist*)malloc(sizeof(dllist));
node2 = (dllist*)malloc(sizeof(dllist));
node3 = (dllist*)malloc(sizeof(dllist));

node1->data='a';
node2->data='b';
node3->data='c';
node1->prev=NULL;
node1->next=node2;
node2->prev=node1;
node2->next=node3;
node3->prev=node2;
node3->next=NULL;
dllist *temp;
for (temp = node1; temp != NULL; temp = temp->next)
   printf("%d ",temp->data);
```



85

Doubly linked list – Example (C proramming)

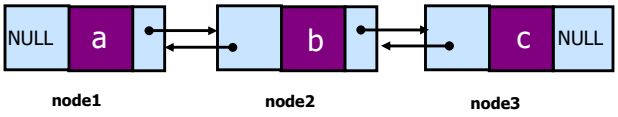
```
typedef struct dllist {
      char data;
      struct dllist *prev;
      struct dllist *next;
}dllist;
dllist *node1, *node2, *node3;
node1 = (dllist*)malloc(sizeof(dllist));
node2 = (dllist*)malloc(sizeof(dllist));
node3 = (dllist*)malloc(sizeof(dllist));

node1->data='a';
node2->data='b';
node3->data='c';
node1->prev=NULL;
node1->next=node2;
node2->prev=node1;
node2->next=node3;
node3->prev=node2;
node3->next=NULL;
dllist *temp;
for (temp = node1; temp != NULL; temp = temp->next)
   printf("%d ",temp->data);
```



86

Doubly linked list – Example (C proramming)

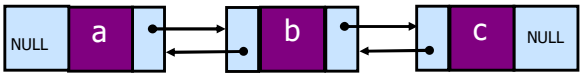
```
typedef struct dllist {
      char data;
      struct dllist *prev;
      struct dllist *next;
}dllist;
dllist *node1, *node2, *node3;
node1 = (dllist*)malloc(sizeof(dllist));
node2 = (dllist*)malloc(sizeof(dllist));
node3 = (dllist*)malloc(sizeof(dllist));

node1->data='a';
node2->data='b';
node3->data='c';
node1->prev=NULL;
node1->next=node2;
node2->prev=node1;
node2->next=node3;
node3->prev=node2;
node3->next=NULL;
dllist *temp;
for (temp = node1; temp != NULL; temp = temp->next)
   printf("%d ",temp->data);
```



87

Doubly linked list – Example (C++)

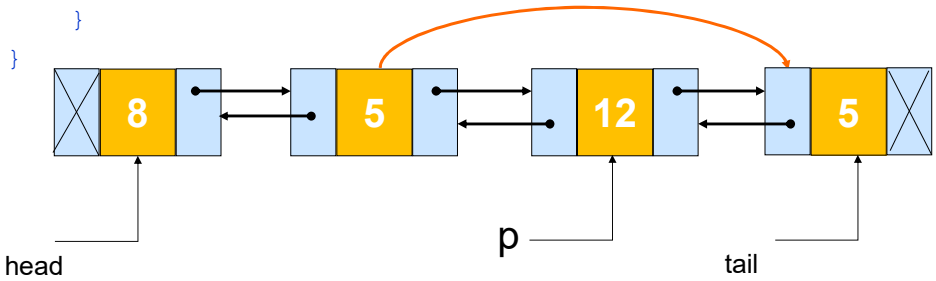
```
typedef struct dllist {
    char data;
    struct dllist *prev;
    struct dllist *next;
}dllist;
dllist node1, node2, node3;

node1.data='a';
node2.data='b';
node3.data='c';
node1.prev=NULL;
node1.next=node2;
node2.prev=node1;
node2.next=node3;
node3.prev=node2;
node3.next=NULL;
dblist *temp;
for (temp = node1; temp != NULL; temp = temp->next)
    cout<<temp->data<<endl;
```



88

Delete a node pointed by the pointer p

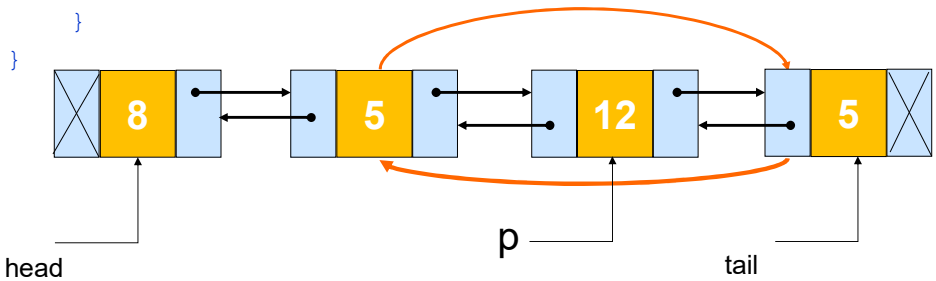
```
void Delete_Node (dllist *p){
    if (head == NULL) printf("Empty list");
    else {
      if (p==head) head = head->next; //Delete first element
      else p->prev->next = p->next;
      if (p->next!=NULL) p->next->prev = p->prev;
      else tail = p->prev;
      free(p);
    }
}
```



89

Delete a node pointed by the pointer p

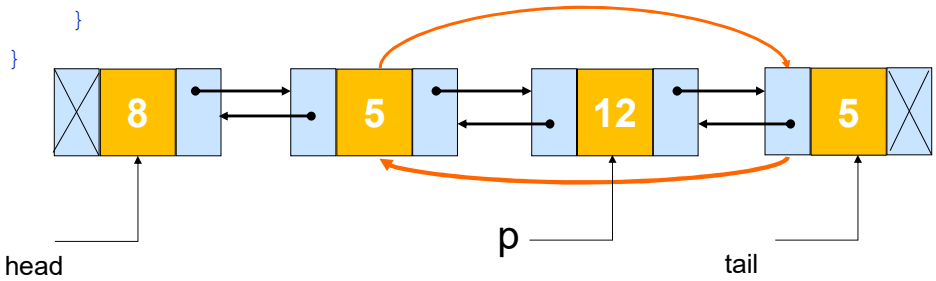
```
void Delete_Node (dllist *p){
    if (head == NULL) printf("Empty list");
    else {
      if (p==head) head = head->next; //Delete first element
      else p->prev->next = p->next;
      if (p->next!=NULL) p->next->prev = p->prev;
      else tail = p->prev;
      free(p);
    }
}
```



90

Delete a node pointed by the pointer p

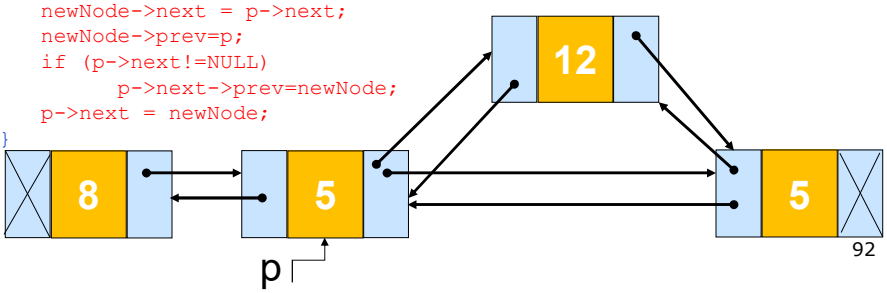
```
void Delete_Node (dllist *p){
    if (head == NULL) printf("Empty list");
    else {
      if (p==head) head = head->next; //Delete first element
      else p->prev->next = p->next;
      if (p->next!=NULL) p->next->prev = p->prev;
      else tail = p->prev;
      free(p);
    }
}
```



91

Insert a node after the node pointed by pointer p

```
void Insert_Node (int X, dllist *p){
   if (head == NULL){  // List is empty
        head =(ddlist*)malloc(sizeof(ddlist));
        head->data = X;
        head->prev =NULL;
        head->next =NULL;
   }
   else{
       dllist *newNode;
       newNode=(dllist*)malloc(sizeof(dllist));
       newNode->data = X;
       newNode->next = NULL;

       newNode->next = p->next;
       newNode->prev=p;
       if (p->next!=NULL)
             p->next->prev=newNode;
       p->next = newNode;
   }
}
```



92

Exercise 2: Create a double linked list store integer numbers

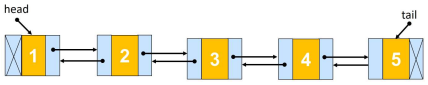
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef struct dllist{
  int number;
  struct dllist *next;
  struct dllist *prev;
} dllist;
dllist *head, *tail;

/* Insert a new node p at the end of the list */
void append_node(dllist *p);
/* Insert a new node p after a node pointed by the pointer after */
void insert_node(dllist *p, dllist *after);
/* Delete a node pointed by the pointer p */
void delete_node(dllist *p);
```

93

```
/* Insert a new node p at the end of the list */
void append_node(dllist *p) {
   if(head == NULL)
   {
      head = p;
      p->prev = NULL;
   }
   else {
     tail->next = p;
     p->prev = tail;
   }
   tail = p;
   p->next = NULL;
}
```

```
/* Insert a new node p after a node pointed by the pointer after */
void insert_node(dllist *p, dllist *after) {
 p->next = after->next;
 p->prev = after;
 if(after->next != NULL)
     after->next->prev = p;
 else
     tail = p;
 after->next = p;
}
/* Delete a node pointed by the pointer p */
void delete_node(dllist *p) {
 if(p->prev == NULL)
     head = p->next;
 else  p->prev->next = p->next;
 if(p->next == NULL)
     tail = p->prev;
 else  p->next->prev = p->prev;
}
```

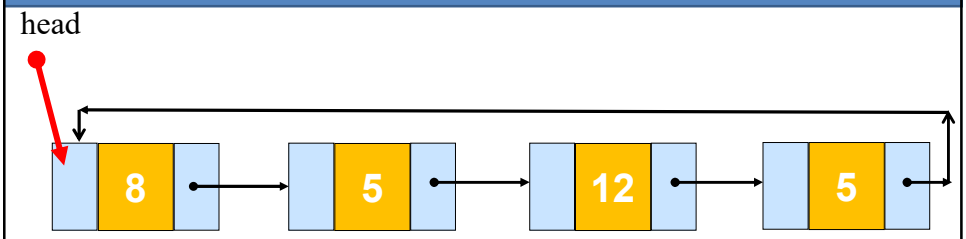
```
 int main( ) {
 dllist *tempnode; int i;
 /* add some numbers to the double linked list */
 for(i = 1; i <= 5; i++) {
    tempnode = (dllist *)malloc(sizeof(dllist));
    tempnode->number = i;
    append_node(tempnode);
 }
/* print the dll list forward */
  printf(" Traverse the dll list forward \n");
  for(tempnode = head; tempnode != NULL; tempnode = tempnode->next)
       printf("%d\n", tempnode->number);

 /* print the dll list backward */
 printf(" Traverse the dll list backward \n");
 for(tempnode = tail; tempnode != NULL; tempnode = tempnode->prev)
       printf("%d\n", tempnode->number);
/* destroy the dll list */
 while(head != NULL) delete_node(head);
 return 0;
}
```



Several variants of linked lists

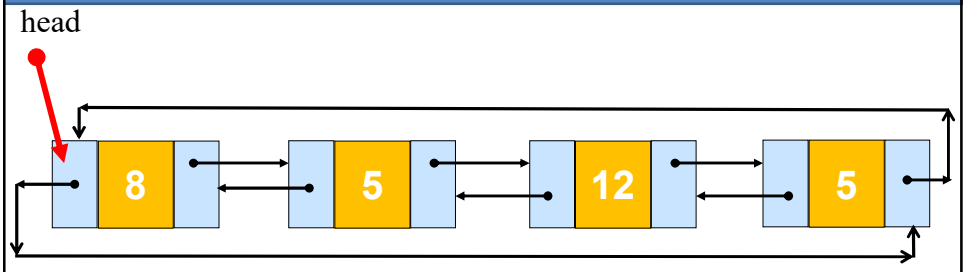
- Some common variants of linked list:
  - Circular Linked Lists
  - Circular Doubly Linked Lists
  - Linked Lists of Lists
- Basic operations on these variants are built similarly to the singly linked list and the doubly linked list that we consider above.

Circular linked list



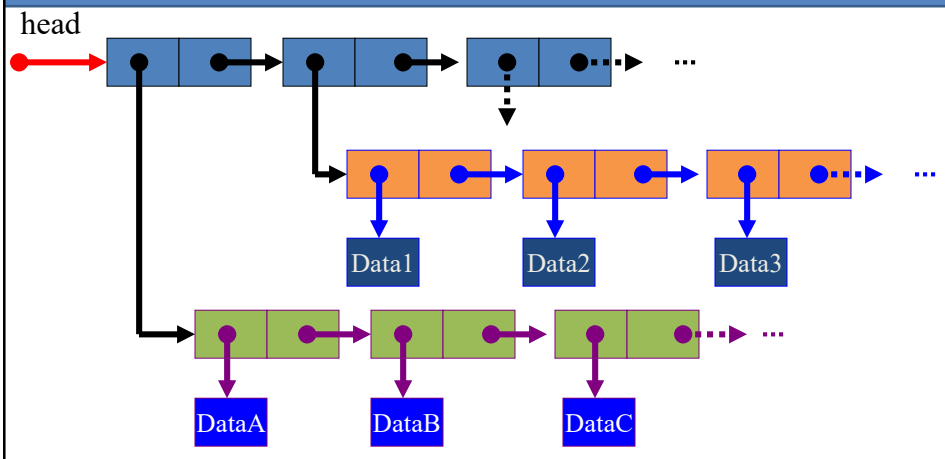

```
typedef struct node{
 int data;
 struct node * next;
}node;
```

Circular Doubly linked list




```
typedef struct node{
 int data;
 struct node * next;
}node;
```

Linked Lists of Lists



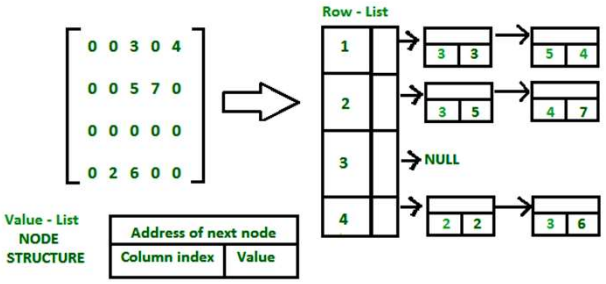
Linked list of lists Application – Sparse matrix

To represent sparse matrix, we could we linked list of list which consisting two lists:

- one list is used to represent the rows and each row contains the list of **triples: Column index, Value(non – zero element) and address field,** for non – zero elements.

For the best performance both lists should be stored in order of ascending keys.

```
// Node to represent triples
typdedef struct value_list
{
    int column_index;
    int value;
    struct value_list *next;
}value_list;

typedef struct row_list
{
    int row_number;
    struct row_list *link_down;
    struct value_list *link_right;
} row_list;
```



Exercise: Polynomial Addition

Polynomials: defined by a list of coefficients and exponents

- *degree* of polynomial = the largest exponent in a polynomial

$$p(x) = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_n x^{e_n}$$

Example:

Polynomials $A(x) = 3x^{10}+2x^5+6x^4+4$

$B(x) = x^4+10x^3+3x^2+1$

To represent a polynomial, the easiest way is to use array $a[i]$ to store the coefficient of x_i.

Operations with polynomials such as: Add two polynomials, Multiply two polynomials, ..., are thus possible to install simply.

Array a ~A(x)	a[10]	a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
	3	0	0	0	0	2	6	0	0	0	4

Array b ~B(x)	b[4]	b[3]	b[2]	b[1]	b[0]
	1	10	3	0	1

$C(x) = A(x) + B(x) = 3x^{10}+2x^5+7x^4+3x^2+5$

Array c ~C(x)	c[10]	c[9]	c[8]	c[7]	c[6]	c[5]	c[4]	c[3]	c[2]	c[1]	c[0]
	=a[10] =3	=a[9] =0	=a[8] =0	=a[7] =0	=a[6] =0	=a[5] =2	=a[4]+b[4] =6+1=7	=a[3]+b[3] =0+10=10	=a[2]+b[2] =0+3=3	=a[1]+b[1] =0+0=0	=a[0]+b[0] =4+1=5

Exercise: Polynomial Addition

$A(x)=2x^{1000}+x^3$
$B(x)=x^4+10x^3+3x^2+1$

Use an array to keep track of the coefficients for all exponents:

…	2	…	0	1	0	0	0	A
…	0	…	1	10	3	0	1	B
	1000	…	4	3	2	1	0	

$A(x) + B(x) =$

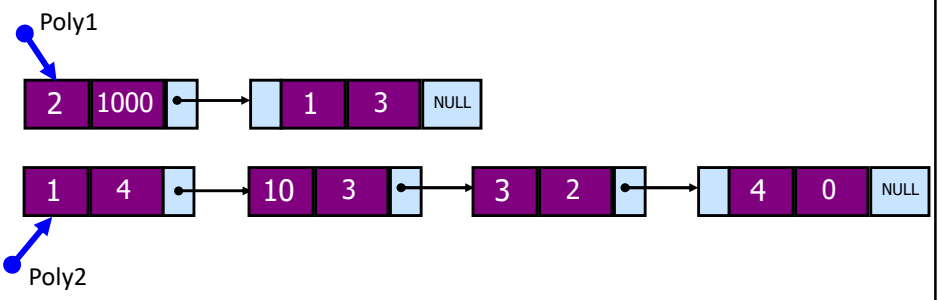
advantage: easy implementation
disadvantage: waste space when sparse

In the case of sparse polynomial (with many coefficients equal to 0), we could represent polynomial by the linked list: We will build a list containing only the coefficients with the value not equal to zero together with the exponent.
However, it is more complicated to implement the operations.

Exercise: Polynomial Addition

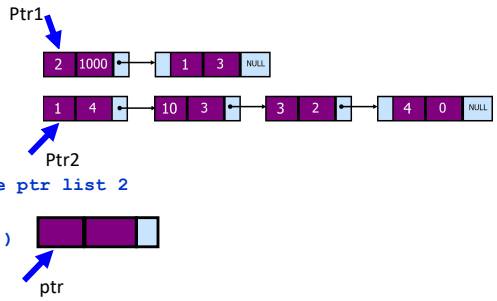
$A(x)=2x^{1000}+x^3$
$B(x)=x^4+10x^3+3x^2+4$

Use the linked list to keep track of the coefficients with values ≠ 0 and the exponent:



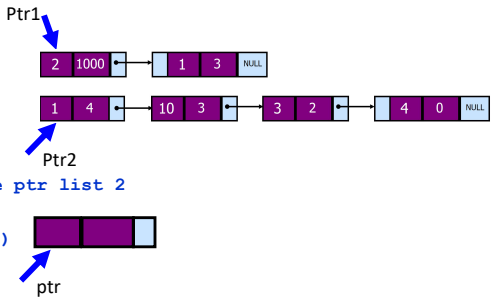

```
typedef struct Polynom {
   int coef;
   int exp;
   struct Polynom *next;
} Polynom;
Polynom *Poly1, *Poly2;
```

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->nextp = node;   //update ptr listResult
} //end of while
```



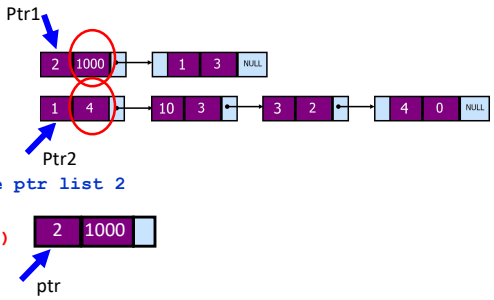

```
if (ptr1 == NULL)      //end of list  1
{
    while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {   node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;   //update ptr list 2
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
     }
 }
 else if (ptr2==NULL)     //end of list  2
 {
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {    node->coef = ptr1->coef;
         node->exp = ptr1->exp;
         ptr1 = ptr1->next;   //update ptr list 2
         ptr = node;
         node = (Polynom *)malloc (sizeof(Polynom));
         ptr->next = node;   //update ptr listResult
    }
 }
 ptr->next = NULL;
}
```

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



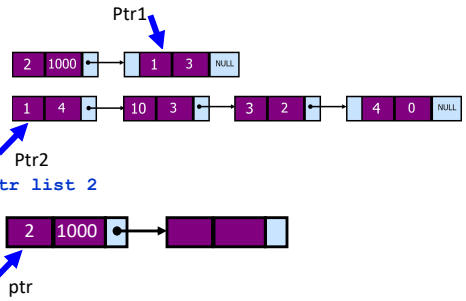
illustration

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



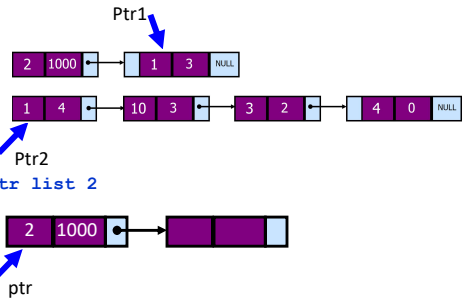
illustration

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



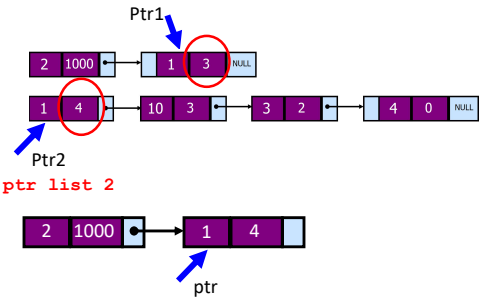
illustration


```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



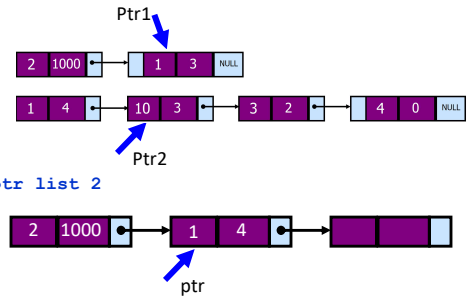
illustration

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



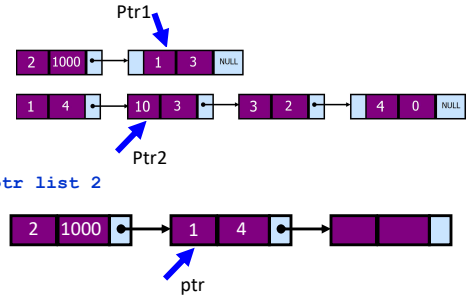
illustration


```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```

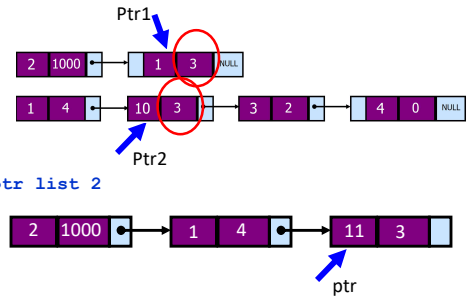
illustration

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



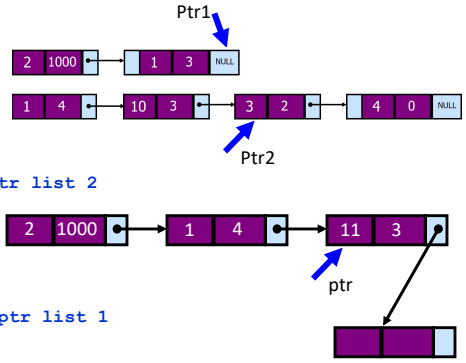
illustration

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



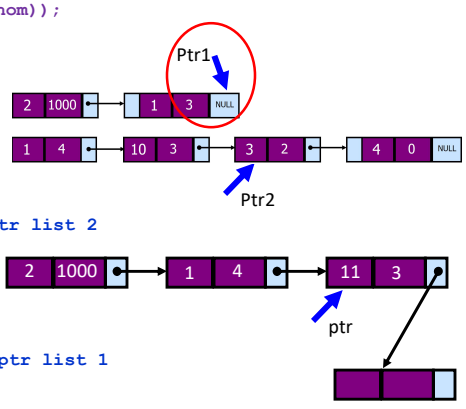
illustration

```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```



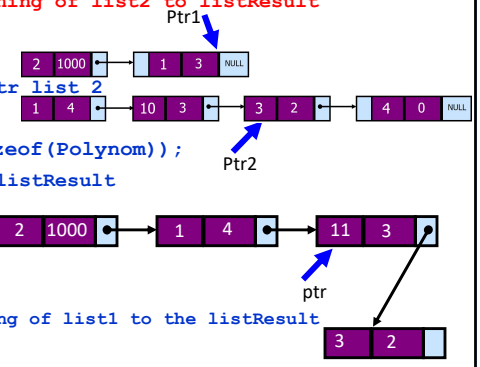
illustration


```
Polynom *PolySum, *node,*ptr,*ptr1,*ptr2;
node = (Polynom *) malloc (sizeof(Polynom));
PolySum = node;
ptr1 = Poly1; ptr2 = Poly2;
while (ptr1!=NULL && ptr2!=NULL){
    ptr = node;
    if  (ptr1->exp < ptr2->exp ) {
       node->coef = ptr2->coef;
       node->exp = ptr2->exp;
       ptr2 = ptr2->next;   //update ptr list 2
    }
    else if ( ptr1->exp > ptr2->exp )
    {   node->coef = ptr1->coef;
        node->exp = ptr1->exp;
        ptr1 = ptr1->next;   //update ptr list 1
    }
    else
    {  node->coef = ptr2->coef + ptr1->coef;
       node->exp = ptr2->exp;
       ptr1 = ptr1->next;   //update ptr list 1
       ptr2 = ptr2->next;   //update ptr list 2
    }
    node = (Polynom *) malloc (sizeof(Polynom));
    ptr->next = node;   //update ptr listResult
} //end of while
```

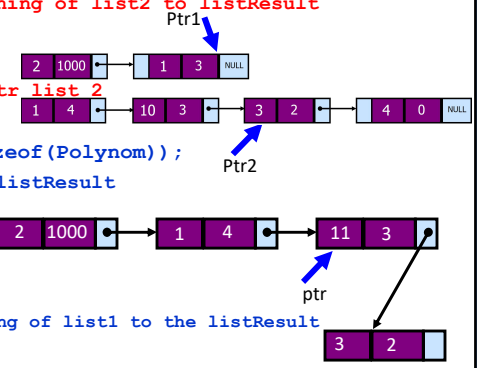
illustration

```
if (ptr1 == NULL)      //end of list  1
{   while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {   node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;   //update ptr list 2
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
     }
 }
 else if (ptr2==NULL)     //end of list  2
 {
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {    node->coef = ptr1->coef;
         node->exp = ptr1->exp;
         ptr1 = ptr1->next;   //update ptr list 2
         ptr = node;
         node = (Polynom *)malloc (sizeof(Polynom));
         ptr->next = node;   //update ptr listResult
    }
 }
 ptr->next = NULL;
}
```



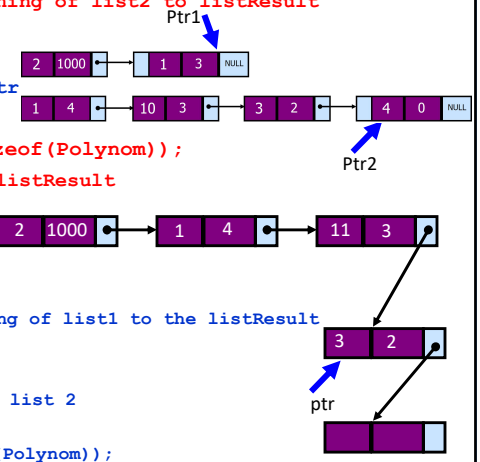
illustration


```
if (ptr1 == NULL)      //end of list  1
{   while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {   node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;   //update ptr list 2
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
     }
 }
 else if (ptr2==NULL)     //end of list  2
 {
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {    node->coef = ptr1->coef;
         node->exp = ptr1->exp;
         ptr1 = ptr1->next;   //update ptr list 2
         ptr = node;
         node = (Polynom *)malloc (sizeof(Polynom));
         ptr->next = node;   //update ptr listResult
    }
 }
 ptr->next = NULL;
}
```



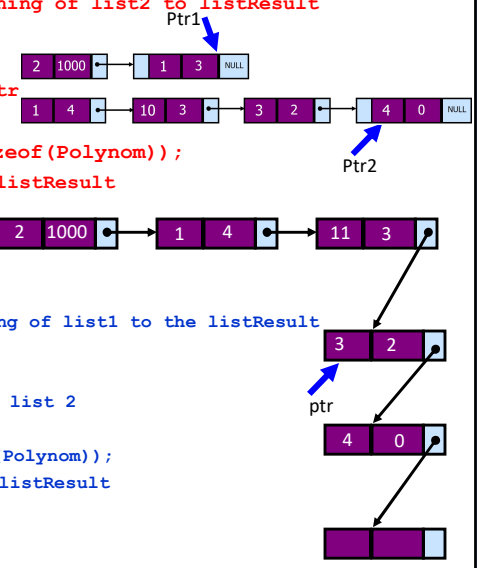
illustration

```
if (ptr1 == NULL)      //end of list  1
{   while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {   node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;   //update ptr
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
     }
 }
 else if (ptr2==NULL)     //end of list  2
 {
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {    node->coef = ptr1->coef;
         node->exp = ptr1->exp;
         ptr1 = ptr1->next;   //update ptr list 2
         ptr = node;
         node = (Polynom *)malloc (sizeof(Polynom));
         ptr->next = node;   //update ptr listResult
    }
 }
 ptr->next = NULL;
}
```



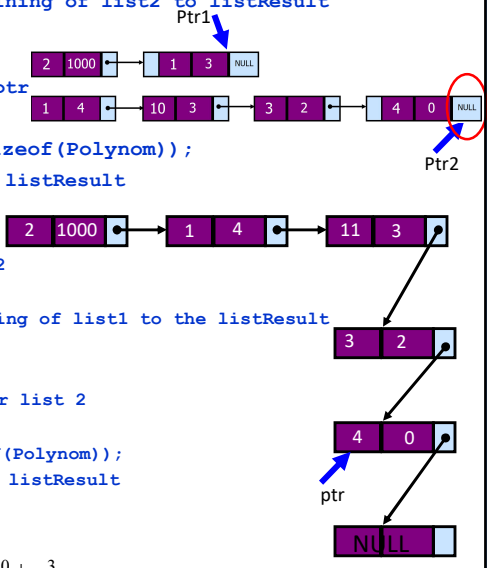
illustration

```
if (ptr1 == NULL)      //end of list  1
{   while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {   node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;   //update ptr
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
     }
 }
 else if (ptr2==NULL)     //end of list  2
 {
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {    node->coef = ptr1->coef;
         node->exp = ptr1->exp;
         ptr1 = ptr1->next;   //update ptr list 2
         ptr = node;
         node = (Polynom *)malloc (sizeof(Polynom));
         ptr->next = node;   //update ptr listResult
    }
 }
 ptr->next = NULL;
}
```



illustration

```
if (ptr1 == NULL)      //end of list  1
{   while(ptr2!= NULL) //copy the remaining of list2 to listResult
    {   node->coef = ptr2->coef;
        node->exp = ptr2->exp;
        ptr2 = ptr2->next;   //update ptr
        ptr = node;
        node = (Polynom *) malloc (sizeof(Polynom));
        ptr->next = node; //update ptr listResult
     }
 }
 else if (ptr2==NULL)     //end of list  2
 {
    while(ptr1 != NULL) //copy the remaining of list1 to the listResult
    {    node->coef = ptr1->coef;
         node->exp = ptr1->exp;
         ptr1 = ptr1->next;   //update ptr list 2
         ptr = node;
         node = (Polynom *)malloc (sizeof(Polynom));
         ptr->next = node;   //update ptr listResult
    }
 }
 ptr->next = NULL;
}
```



$A(x) = 2x^{1000} + x^3$

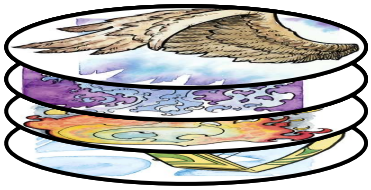
$B(x) = x^4 + 10x^3 + 3x^2 + 4$

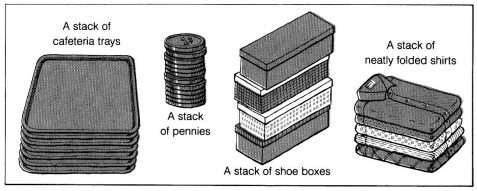
$Result(x) = 2x^{1000} + x^4 + 11x^3 + 3x^2 + 4$

Contents

What is a stack?

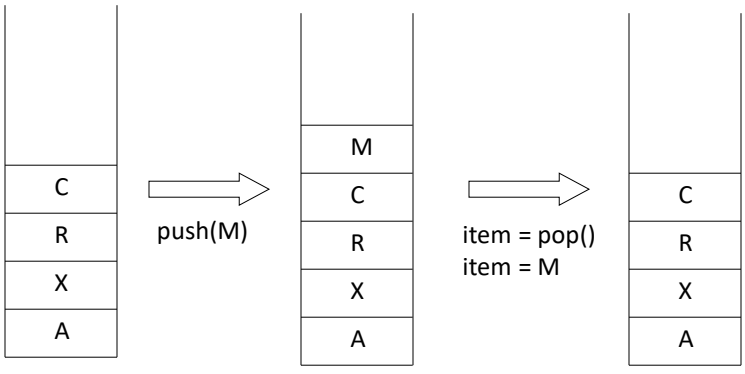
- A stack is a data structure that only allows items to be inserted and removed at ***one end***
  - We call this end the ***top*** of the stack
  - The other end is called the bottom
- Access to other items in the stack is not allowed
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out)





Operation on stack

- *Push*: the operation to place a new item at the top of the stack
- *Pop:* the operation to remove the next item from the top of the stack

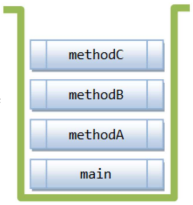


What Are Stacks Used For?

- Real life (Pile of books, Plate trays, etc.)
- More applications related to computer science
  - Program execution stack: Most programming languages use a "call stack" to implement function calling
    - When a method is called, its line number and other useful information are pushed (inserted) on the call stack
    - When a method ends, it is popped (removed) from the call stack and execution restarts at the indicated line number in the method that is now at the top of the stack

```
void methodC()
{
    printf("Enter methodC ");
}
void methodB()
{
    methodC();
}
void methodA()
{
    methodB();
}
void main() {
    methodA();
}
```

1. main pushed onto call stack, before invoking methodA
2. methodA pushed onto call stack, before invoking methodB
3. methodB pushed onto call stack, before invoking methodC
4. methodC pushed onto call stack, invoked then popped out from the call stack when completes
5. methodB popped out from call stack when completes.
6. methodA popped out from the call stack when completes.
7. main popped out from the call stack when completes. Program exits.



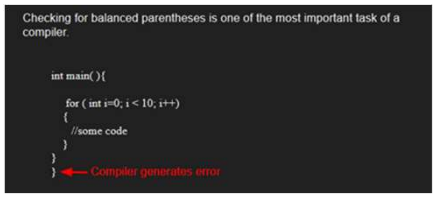
Method Call Stack
(Last-in-First-out Queue)

  - Evaluating expressions (e.g. (4/(2-2+3))*(3-4)*2)

What Are Stacks Used For?

- More applications related to computer science
  - compilers
    - parsing data between delimiters (brackets)
    - Check: each "(", "{", or "[" has to pair with ")", "}", or "]"

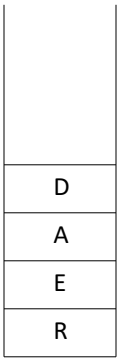
    Example:
      - correct: ( )(( )){([( )])}
      - correct: ((( )(( )){([( )])}
      - incorrect:)(( )){([( )])}
      - incorrect: ({[ ])}
      - incorrect: (



  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - artificial intelligence
    - finding a path

Example: Reversing a Word

- We can use a stack to reverse the letters in a word.
- How?
- Example: READ



Push(R)

Push(E)

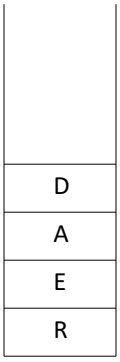
Push(A)

Push(D)

- Read each letter in the word and push it onto the stack

Example: Reversing a Word

- We can use a stack to reverse the letters in a word.
- How?
- Example: READ



Push(R) Pop(D)

Push(E) Pop(A)

Push(A) Pop(E)

Push(D) Pop(R)

DAE R

- Read each letter in the word and push it onto the stack
- When you reach the end of the word, pop the letters off the stack and print them out

Implementing a Stack

- At least two different ways to implement a stack:
  - array
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

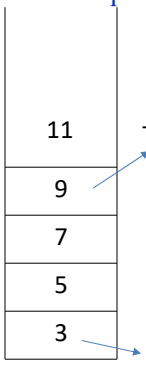
Stack: Array Implementation

- A stack consists of 4 elements: 3, 5, 7, 9.
- If an array is used to implement this stack, what is a good index for the top item?
  - Is it position 0?

S[0]=9	S[1]=7	S[2]=5	S[3]=3

  - Is it position 3?

S[0]=3	S[1]=5	S[2]=7	S[3]=9



Let's think when we need do operation:
- PUSH
- POP

The bottom of the stack is at `S[0]`
The top of the stack is at `S[3]`
- *pop* off of the stack at `S[3]`
- *push* onto the stack at `S[4]`

Stack: Array Implementation

- If an array is used to implement a stack, the index for the top item is `numItems-1` (where `numItems` is the number of items in the stack)
- Note that push and pop must both work in O(1) time as stacks are usually assumed to be extremely fast
- Implementing a stack using an array is fairly easy:
  - The bottom of the stack is at `S[0]`
  - The top of the stack is at `S[numItems-1]`
  - *push* onto the stack at `S[numItems]`
  - *pop* off of the stack at `S[numItems-1]`

Example Stack: Array Implementation

Basic operations:

- **void STACKinit(int);**
- **int STACKempty();**
- **void STACKpush(Item);**
- **Item STACKpop();**

```
static int *s;
static int maxSize;//maximum number of elements that the stack could have
static int numItems; //current number of elements on stack
void STACKinit(int maxSize)
{
    s = (int *) malloc(maxSize*sizeof(int));
    N = 0;
}
int STACKempty(){return numItems==0;}
int STACKfull() {return numItems==maxSize;}
```

```
void STACKpush(int item)
{
   if (Stackfull()) ERROR("Stack is full");
   else
   {   s[numItems] = item;
       numItems++;
   }
}
Item STACKpop()
{
   if (STACKempty()) ERROR("Stack is empty")
   else
   {
      numItems--;
      return s[numItems+1];
   }
}
```

Array Implementation Summary

- Advantages
  - Easy to implement
  - best performance: push and pop can be performed in O(1) time
- Disadvantage
  - fixed size: the size of the array must be initially specified because
    - The array size must be known when the array is created and is fixed, so that the right amount of memory can be reserved
    - Once the array is full no new items can be inserted
- If the maximum size of the stack is not known (or is much larger than the expected size) a dynamic array can be used
  - But occasionally push will take O(n) time

maxSize: maximum number of elements in the array

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(STACKfull())
        STACKpush(item);
```

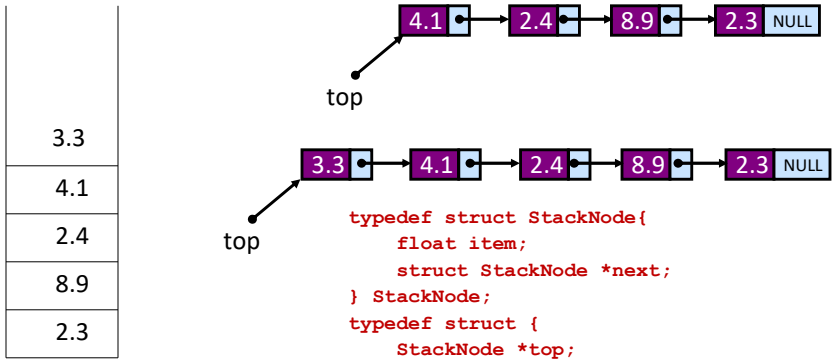
Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if (STACKempty())
      STACKpop(item);
```

Implementing a Stack: using linked list

- Store the items in the stack in a linked list
- The top of the stack is the head node, the bottom of the stack is the end of the list
- *push* by adding to the front of the list
- *pop* by removing from the front of the list




```
typedef struct StackNode{
    float item;
    struct StackNode *next;
} StackNode;
typedef struct {
    StackNode *top;
}Stack;
```

Operations

1. Init:

```
Stack *StackConstruct();
```

2. Check empty:

```
int StackEmpty(Stack* s);
```

3. Check full:

```
int StackFull(Stack* s);
```

4. Insert a new item into stack (Push): *insert a new item at the top of stack*

```
int StackPush(Stack* s, float* item);
```

5. Remove an item from stack (Pop): *remove and return the item at the top of stack:*

```
float pop(Stack* s);
```

6. Print out all items of stack

```
void Disp(Stack* s);
```

Initialize stack

```
Stack *StackConstruct() {
    Stack *s;
    s = (Stack *)malloc(sizeof(Stack));
    if (s == NULL) {
      return NULL;  // No memory
    }
    s->top = NULL;
    return s;
}

/****   Destroy stack   *****/
void StackDestroy(Stack *s) {
    while (!StackEmpty(s)) {
      StackPop(s);
    }
    free(s);
}
```

137

```
/*** Check empty   ***/
int StackEmpty(const Stack *s) {
    return (s->top == NULL);
}

/*** Check full  ***/
int StackFull() {
    printf("\n NO MEMORY! STACK IS FULL");
    return 1;
}
```

138

Display all items in the stack

```
void disp(Stack* s) {
   StackNode* node;
   int ct = 0; float m;
   printf("\n\n  List of all items in the stack \n\n");
   if (StackEmpty(s))
       printf("\n\n  >>>>>  EMPTY STACK   <<<<<\n");
   else {
      node = s->top;
      do {
         m = node->item;
         printf("%8.3f \n", m);
         node = node->next;
       } while (!(node == NULL));
   }
}
```

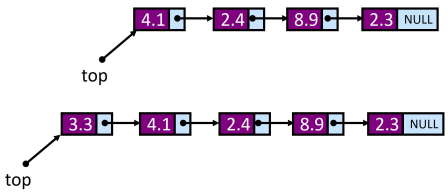
Push

Need to do the following steps:
(1) Create new node: allocate memory and assign data for new node
(2) Link this new node to the top (head) node
(3) Assign this new node as top (head) node

```
int StackPush(Stack *s, float item) {
  StackNode *node;
  node = (StackNode *)malloc(sizeof(StackNode)); //(1)
  if (node == NULL) {
    StackFull(); return 1;  // overflow: out of memory
  }
  node->item = item;        //(1)
  node->next = s->top;      //(2)
  s->top = node;            //(3)
  return 0;
}
```

Pop

1. Check whether the stack is empty
2. Memorize address of the current top (head) node
3. Memorize data of the current top (head) node
4. Update the top (head) node: the top (head) node now points to its next node
5. Free the old top (head) node
6. Return data of the old top (head) node




```
float StackPop(Stack *s) {
    float data;
    StackNode *node;
    if (StackEmpty(s))          //(1)
      return NULL;          // Empty Stack, can't pop
    node = s->top;              //(2)
    data = node->item;          //(3)
    s->top = node->next;        //(4)
    free(node);                 //(5)
    return item;                //(6)
}
```

Experimental program

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
// all above functions of stacks are put here
int main() {
   int ch,n,i;   float m;
   Stack* stackPtr;
   while(1)
   {  printf("\n\n======================\n");
      printf(" STACK TEST PROGRAM \n");
      printf("======================\n");
      printf(" 1.Create\n 2.Push\n 3.Pop\n 4.Display\n 5.Exit\n");
      printf("----------------------\n");
      printf("Input number to select the appropriate operation: ");
      scanf("%d",&ch); printf("\n\n");
```

142

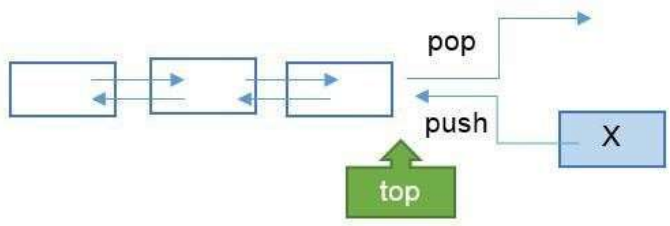
```
  switch(ch) {
     case 1:    printf("INIT STACK");
                stackPtr = StackConstruct(); break;
     case 2:    printf("Input float number to insert into stack: ");
                scanf("%f",&m);
                StackPush(stackPtr, m); break;
     case 3:    m=StackPop(stackPtr);
                if (m != NULL)
                  printf("\n Data Value of the popped node: %8.3f\n",m);
                else {
                  printf("\n  >>> Empty Stack, can't pop <<<\n");}
                break;
     case 4:    disp(stackPtr); break;
     case 5:    printf("\n  Bye! Bye! \n\n");
                exit(0);    break;
     default:   printf("Wrong choice");
   } //switch
  } // end while
} //end main
```


Implementing a Stack: using linked list

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - Difficult to implement

Library STL in C++: stack

`#include <stack>`: a list type where insertion and removal are both performed at the end of the list.



Example about declaration:

```
stack <int> s1; //declare variable stack s1 in which type of each element in s1 is int
```

- We can not declare `stack<int> myStack (5)` to initialize a stack with 5 empty elements as in array.
- Similarly, we can not declare `stack<int> myStack (5,100)` to initialize a stack consisting of 5 elements with values 100.

Library STL in C++: stack

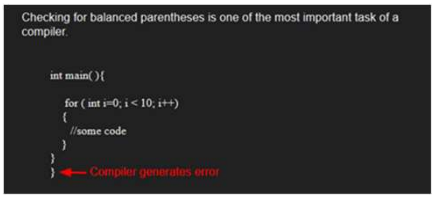
Capacity	
size()	Return the number of elements currently in stack
empty()	True/False
Acess to an element	
top()	Acess to top element of stack
Query	
push(x)	Insert an element with value x on the top of stack. Size of stack will be increased by 1
pop()	Remove the top element from stack
swap(stack s1, stack s2)	Swap elements of stack s1 and s2

Application 1: Parentheses Matching

Check for balanced parentheses in an expression:

Given an expression string expression, write a program to examine whether the pairs and the orders of "{","}","(",")","[","]" are correct in expression.

For example, the program should print true for expression = "[()]{}{[()()]()}" and false for expression = "[(])"



Algorithm:

1) Declare a character stack S.

2) Now traverse the expression string expression

a) If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.

b) If the current character is a closing bracket (')' or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.

3) After complete traversal, if there is some starting bracket left in stack then "not balanced"

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Application 1: Parentheses Matching

Algorithm ParenMatch(X,n):

Input: Array X consists of n characters, each character could be either parentheses, variable, arithmetic operation, number.

Output: **true if** parentheses in an expression are **balanced**

```
S = stack empty;
for i=0 to n-1 do
    if (X[i] is a starting bracket)
        push(S, X[i]);   // starting bracket ('(' or '{' or '[') then push it to stack
    else
        if (X[i] is a closing bracket)  // compare X[i] with the one currently on the top of stack
            if isEmpty(S)
                return false {can not find pair of brackets}
            if (pop(S) not pair with bracket stored in X[i])
                return false {error: type of brackets}
if isEmpty(S)
    return true {parentheses are balanced}
else  return false {there exist a bracket not paired}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Application 2: Calculate the value of an expression

Example: Calculate the value of expression: (4/(2-2+3))*(3-4)*2)

Node: Priority order of operations from high to low:

1. Exponent ^
2. Multiplication Division * /
3. Addition subtraction + -

The algorithm use 2 stacks:

- Stack S1 stores operands
- Stack S2 stores operators (+,-,*,/, ^) and opening bracket (

Procedure **Process**:

- Do Pop(S1) twice: Pop off 2 values on the top of S1 from S1, denote these 2 values by **A** and **B**
- Do Pop(S2): Pop off 1 operator on the top of S2 from S2, denote it by **OPT**.
- Do **B OPT A**, the result is stored in **R**
- Push R into S1: do Push(S1, R)

Application 2: Calculate the value of an expression

Scan the expression from left to right:

- If the current element is *Operand*: push it into stack S1.
- If the current element if *Operator, denote it by **OPT***:
  - If stack S2 is empty: push operator **OPT** into stack S2, that means: Push(S2, OPT)
  - Else if **OPT** has higher priority than that of top(S2) then push OPT into stack S2, that means Push(S2, OPT)
  - Else if **OPT** has priotiry lower or equal to that of top(S2) then
    - do Process Until operator OPT has higer priotiy than that of top(S2) OR stack S1 is empty
    - Push(S2, OPT)
- If the current element if *opening bracket:* then push it into stack S2.
- If the current element if *closing bracket*: then
  - do Process … Until found the first opening bracket in S2
  - Pop the opening bracket from S2
- *When finish to scan the expression and Stack S1 is not empty*:
  - do Process Until Stack S1 is empty

Example: Calculate 2 * (5 * (3 + 6)) / 15 - 2

Element	Operation	Stack S1	Stack S2	Explanation
2	Push 2 into S1	2	Empty	
*	Push * into S2	2	*	
(	Push (into S2	2	(*	
5	Push 5 into S1	5 2	(*	
*	Push * into S2	5 2	* (*	
(	Push (into S2	5 2	(* (*	
3	Push 3 into S1	3 5 2	(* (*	
+	Push + into S2	3 5 2	+ (* (*	
6	Push 6 into S1	6 3 5 2	+ (* (*	

Scan the expression from left to right:

- If the current element is *Operand*: push it into stack S1.
- If the current element if *Operator, denote it by **OPT***:
  - If stack S2 is empty: Push(S2, OPT)
  - Else if **OPT** has higher priority than that of top(S2) then Push(S2, OPT)
  - Else if **OPT** has priotiry lower or equal to that of top(S2) then
    - do Process Until operator **OPT** has higer priotiy than that of top(S2) OR stack S1 is empty
    - Push(S2, OPT)
- If the current element if *opening bracket:* then push it into stack S2.
- If the current element if *closing bracket*: then
  - do Process … Until found the first opening bracket in S2
  - Pop the opening bracket from S2
- *When finish to scan the expression and Stack S1 is not empty*:
  - do Process Until Stack S1 is empty

Example: Calculate 2 * (5 * (3 + 6)) / 15 - 2

Procedure **Process**:

- Do Pop(S1) twice: Pop off 2 values on the top of S1 from S1, denote these 2 values by **A** and **B**
- Do Pop(S2): Pop off 1 operator on the top of S2 from S2, denote it by **OPT**.
- Do **B OPT A**, the result is stored in **R**
- Push R into S1: do Push(S1, R)

)	Call Process:			Do Process until found the first opening bracket in S2
	Pop 6 and 3 from S1	5 2	+ (* (*	
	Pop + from S2	5 2	(* (*	
	Do 3+6=9	5 2	(* (*	
	Push 9 into S1	9 5 2	(* (*	
	Pop (from S2	9 5 2	* (*	

Example: Calculate 2 * (5 * (3 + 6)) / 15 - 2

Element	Operation	Stack S1	Stack S2	Explanation
)	Call Process:			Do Process until found the first opening bracket in S2
	Pop 9 and 5 from S1	2	* (*	
	Pop * from S2	2	(*	
	Do 5*9=45	2	(*	
	Push 45 into S1	45 2	(*	
	Pop (from S2		*	
/	Call Process:			/ and * have the same priority Do Process
	Pop 45 and 2 from S1		*	
	Pop * from S2			
	Do 2*45=90			
	Push 90 into S1	90		
	Push / into S2		/	

Scan the expression from left to right:

- If the current element is *Operand*: push it into stack S1.
- If the current element if *Operator, denote it by **OPT***:
  - If stack S2 is empty: Push(S2, OPT)
  - Else if **OPT** has higher priority than that of top(S2) then Push(S2, OPT)
  - Else if **OPT** has priotiry lower or equal to that of top(S2) then
    - do Process Until operator **OPT** has higer priotiy than that of top(S2) OR stack S1 is empty
    - Push(S2, OPT)
- If the current element if *opening bracket:* then push it into stack S2.
- If the current element if *closing bracket*: then
  - do Process … Until found the first opening bracket in S2
  - Pop the opening bracket from S2
- *When finish to scan the expression and Stack S1 is not empty*:
  - do Process Until Stack S1 is empty

Procedure **Process**:

- Do Pop(S1) twice: Pop off 2 values on the top of S1 from S1, denote these 2 values by **A** and **B**
- Do Pop(S2): Pop off 1 operator on the top of S2 from S2, denote it by **OPT**.
- Do **B OPT A**, the result is stored in **R**
- Push R into S1: do Push(S1, R)

Example: Calculate 2 * (5 * (3 + 6)) / 15 - 2

Element	Operation	Stack S1	Stack S2	Explanation
15	Push 15 into S1	15 90	/	
-	Call Process		/	"-" has lower priority than "/", so do Process
	Pop 15 and 90 from S1			
	Pop / from S2			
	Do 90/15=6			
	Push 6 into S1	6		
-	Push - into S2	6	-	

Scan the expression from left to right:

- If the current element is *Operand*: push it into stack S1.
- If the current element if *Operator, denote it by **OPT***:
  - If stack S2 is empty: Push(S2, OPT)
  - Else if **OPT** has higher priority than that of top(S2) then Push(S2, OPT)
  - Else if **OPT** has priotiry lower or equal to that of top(S2) then
    - do Process Until operator **OPT** has higer priotiy than that of top(S2) OR stack S1 is empty
    - Push(S2, OPT)
- If the current element if *opening bracket:* then push it into stack S2.
- If the current element if *closing bracket*: then
  - do Process … Until found the first opening bracket in S2
  - Pop the opening bracket from S2
- *When finish to scan the expression and Stack S1 is not empty*:
  - do Process Until Stack S1 is empty

Procedure **Process**:

- Do Pop(S1) twice: Pop off 2 values on the top of S1 from S1, denote these 2 values by **A** and **B**
- Do Pop(S2): Pop off 1 operator on the top of S2 from S2, denote it by **OPT**.
- Do **B OPT A**, the result is stored in **R**
- Push R into S1: do Push(S1, R)

Example: Calculate 2 * (5 * (3 + 6)) / 15 - 2

Element	Operation	Stack S1	Stack S2	Explanation
-	Push - into S2	6	-	
2	Push 2 into S1	2 6	-	
	Call Process			Already finish to scan all elements in expression, but S1 is not empty. Therefore, call Process until S2 is empty
	Pop 2 and 6 from S1	EMPTY	-	
	Pop – from S2	EMPTY	EMPTY	
	Do 6-2=4			
	Push 4 into S1	4		

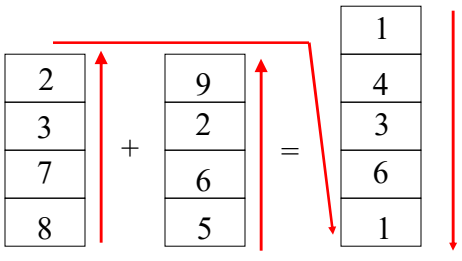
The value of expression = 4

Application 3: Adding two large numbers

- Treat these numbers as strings of numerals, store the numbers corresponding to these numerals on two stacks, and then perform addition by popping numbers from the stacks

12345678901234567890123456789012345

78901234567890123456789012345



8732 + 5629 = 14361

Application 3: Adding two large numbers

- Read the numerals of the first number and store the numbers corresponding to them on one stack;
- Read the numerals of the second number and store the numbers corresponding to them on another stack;
- **carry**=0;
- while at least one stack is not empty
  - pop a number from each non-empty stack, add them and **carry**;
  - push the sum (minus 10 if necessary) on the result stack;
  - store carry in **carry** variable;
- push carry on the result stack if it is not zero;
- pop numbers from the result stack and display them

157

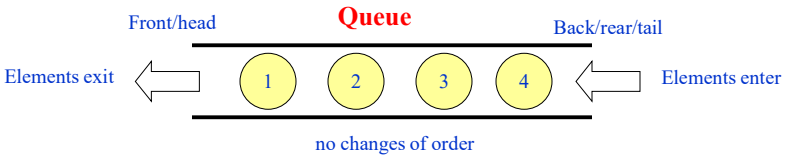
Contents

158

What is a Queue?

Queues

- What is a queue?
  - A data structure of ordered items such that items can be inserted only at one end and removed at the other end.



Example: A line at the supermarket

- What can we do with a queue?
  - **Enqueue** (insert) – Add an item to the queue (also called as insert)
  - **Dequeue** (delete) – Remove an item from the queue (also called as getFront)

- A queue is called a FIFO (First in-First out) data structure.

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Queue

- Terms used in Queue are depicted in the following figure:

NGUYỄN KHÁNH PHƯƠNG
KHMT – SOICT - ĐHBK HN

Queue specification

Definitions: (provided by the user)

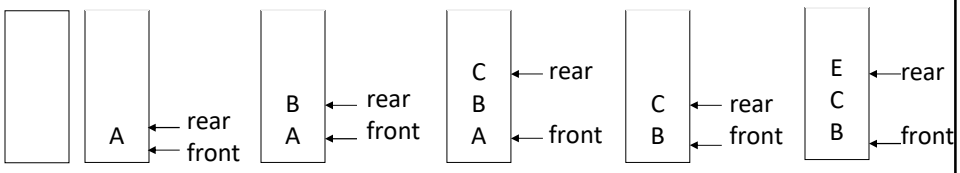
- *maxSize:* Max number of items that might be on the queue
- *ItemType*: Data type of the items on the queue

Operations:

- `Q = init();` initialize empty queue Q
- `isEmpty(Q);` returns "true" if queue Q is empty
- `isFull(Q);` returns "true" if Q is full, indicates that we already use the maximum memory for queue; otherwise returns "false"
- `frontQ(Q);` returns the item that is in front (head) of queue Q or returns error if queue Q is empty.
- `enqueue(Q,x);` inserts item x into the back (rear) of queue Q. If before making insertion, the queue Q is full, then give the notification about that.
- `x = dequeue(Q);` deletes the element at the front (head) of the queue Q, then returns x which is the data of this element. If the queue Q is empty before dequeue, then give the error notification.
- `print(Q);` gives the list of all elements in the queue Q in the order from the front to the back.
- `sizeQ(Q);` returns the number of elements currently in the queue Q.

FIFO

Back/Rear Front/Head



enqueue(Q, A) enqueue(Q, B) enqueue(Q, C) dequeue(Q, A) enqueue(Q, E)

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Queues everywhere!!!!

Queues

- What are some applications of queues?
  - Round-robin scheduling in processors
  - Input/Output processing
  - Queueing of packets for delivery in networks

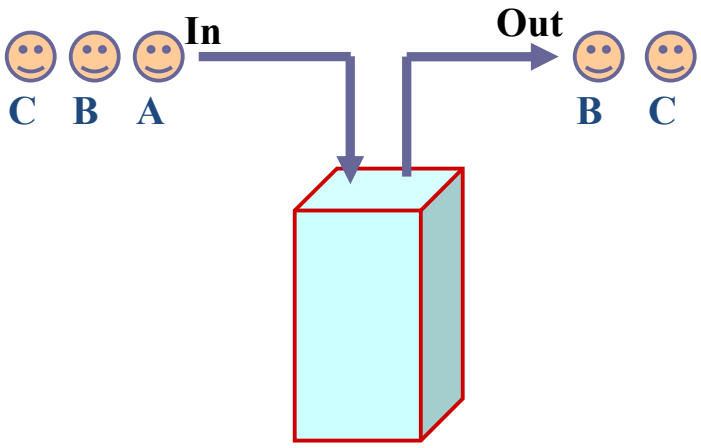
NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Given the sequence of operations on queue Q as following. Determine the output and the data on the queue Q after each operation:

	Operation	Output	Queue Q
1	`enqueue(5)`	-	(5)
2	`enqueue(Q,3)`	-	(5, 3)
3	`dequeue(Q)`	5	(3)
4	`enqueue(Q,7)`	-	(3, 7)
5	`dequeue(Q)`	3	(7)
6	`front(Q)`	7	(7)
7	`dequeue(Q)`	7	( )
8	`dequeue(Q)`	error	( )
9	`isEmpty(Q)`	true	( )
10	`size(Q)`	0	( )
11	`enqueue(Q,9)`	-	(9)
12	`enqueue(Q,7)`	-	(9, 7)
13	`enqueue(Q,3)`	-	(9, 7, 3)
14	`enqueue(Q,5)`	-	(9, 7, 3, 5)
15	`dequeue(Q)`	9	(7, 3, 5)

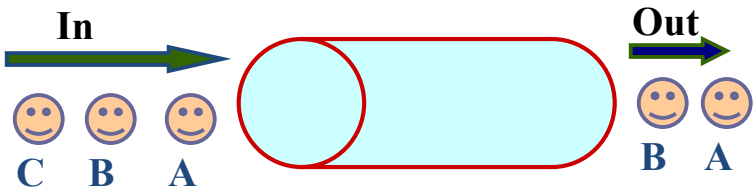
Stack

Data structure with Last-In First-Out (LIFO) behavior



Queue

Data structure with First-In First-Out (FIFO) behavior

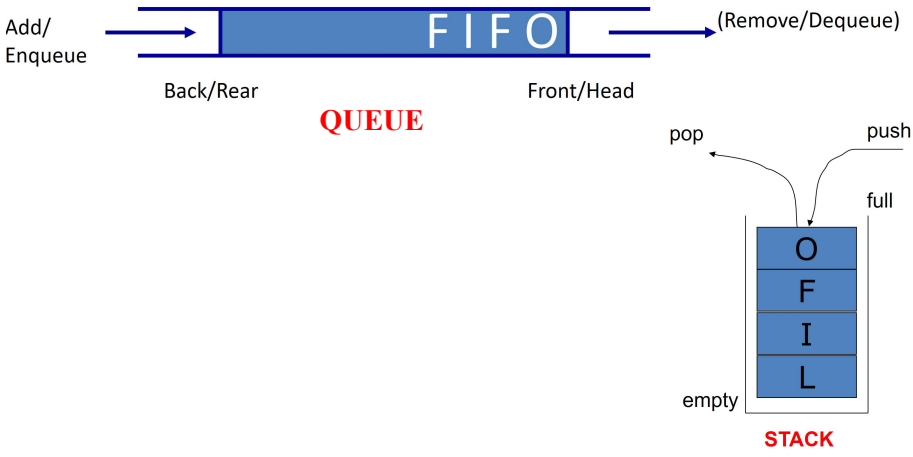


Implementing a Queue

- Just like a stack, we can implementing a queue in two ways:
  - Using an array
  - Using a linked list

Implementing a Queue: using Array

- Using an array to implement a queue is significantly harder than using an array to implement a stack. Why?
  - A stack: we add and remove at the same end,
  - A queue: we add to one end and remove from the other.

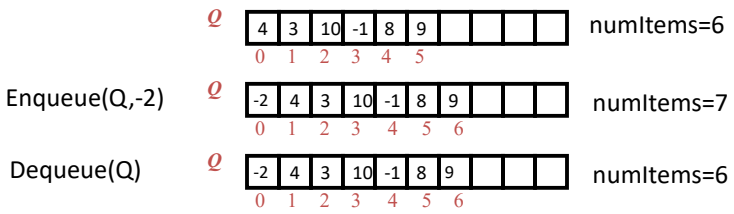


Implementing a Queue: using Array

There are some options for implementing a queue using an array:

- Option 1:
  - *Enqueue* at Q[0] and shift all of the rest of the items in the array down to make room.
  - *Dequeue* from Q[numItems-1]

Example:



- Option 2: circular queue ["wrap around"]

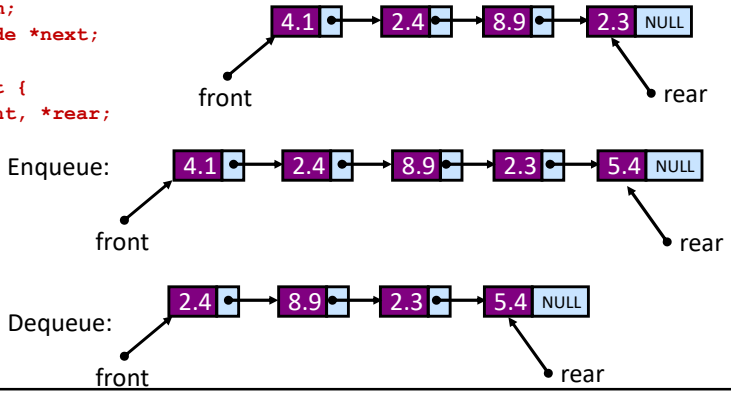
Implementing a Queue

- Just like a stack, we can implementing a queue in two ways:
  - Using an array
  - **Using a linked list**

Implementing a Queue: using linked list

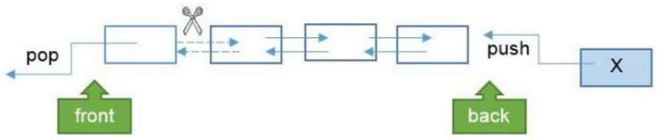
- Store the items in the queue in a linked list
- The top of the queue is the head node, the bottom of the queue is the end of the list
- *Enqueue* by adding a new element to the front of the list
- *Dequeue* by removing the last element from the list

```
typedef struct node{
    float item;
    struct node *next;
}node;
typedef struct {
    node *front, *rear;
}Queue;
```



Library STL in C++: queue

- `#include <queue>` a list type where insertion is performed at the end of the list and removal at the top of the list.



Example of declaration:

`queue <int> q1;` //declare queue q1 in which type of each element is `int`

- Similar to stack, it is not allowed to declare *`queue<int> myQueue (5)`* to initialize 5 empty elements.
- Similarly, it is not allowed to decalre *`queue<int> myQueue (5,100)`* to initialize 5 elements in queue with value of 100.

Library STL in C++: queue

Capacity	
size()	Return number of elements in the queue
empty()	True/False
Access to an element	
front()	Access to the first element of the queue
back()	Access to the last element of the queue
Query	
push(x)	Insert a new element of value x at the end of the queue. Size of queue will be increased by 1
pop()	Remove the element at the top of queue. Size of queue will be reduced by 1
swap(queue q1, queue q2)	Swap elements of queue q1 and q2

Example

```
#include <iostream>
#include <queue>
using namespace std;
int main ()
{
  queue <int> myqueue;
  int myint;
  cout << "Please enter some integers (enter 0 to end):\n";
  do {
    cin >> myint;
    myqueue.push (myint);
  } while (myint);
  cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
       cout << " " << myqueue.front();
       myqueue.pop();
  }
  return 0;
}
```

Application 1: recognizing palindromes

- A *palindrome* is a string that reads the same forward and backward.

Example: NOON, DEED, RADAR, MADAM

Able was I ere I saw Elba

- How to recognize a given string is a palindrome or not:
  - Step 1: We will put all characters of the string into both a stack and a queue.
  - Step 2: Compare the contents of the stack and the queue character-by-character to see if they would produce the same string of characters:
    - If yes: the given string is a palindrome
    - Otherwise: not palindrome

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example 1: Whether "RADAR" is a palindrome or not

Step 1: Put "RADAR" into Queue and Stack:

Current character	Queue (front on the left, rear on the right)	Stack (top on the left)
R	R	R
A	R A	A R
D	R A D	D A R
A	R A D A	A D A R
R	R A D A R	R A D A R

front rear top

Example 1: Whether “RADAR” is a palindrome or not

Step 2: Delete “RADAR” from Queue and Stack:

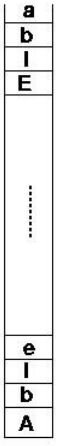
- Dequeue until the queue is empty
- Pop the stack until the stack is empty

Queue (front on the left)	Front of Queue	Top of Stack	Stack (top on the left)
R A D A R	R	R	R A D A R
A D A R	A	A	A D A R
D A R	D	D	D A R
A R	A	A	A R
R	R	R	R
empty	empty	empty	empty

Conclusion: String "RADAR" is a palindrome

Example 2: recognizing palindromes

Able was I ere I saw Elba



Stack

Queue

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Application 2: Convert a string of digits into a decimal number

The algorithm is described as following:

```
// Convert sequence of digits stored in queue Q into decimal number n
// Remove empty space if any
do {      dequeue(Q, ch)
} until ( ch != blank)
//  ch is now the first digit of the given string
// Calculate n from sequence of digit in the queue
n = 0;
done = false;
do {      n = 10 * n + decimal number that ch represents;
          if (! isEmpty(Q) )
                    dequeue(Q,ch)
          else
                    done = true
}  until ( done  || ch != digit)
// Result: n is the decimal number need to be found
```