



РОЗШИРЕНЕ ПРЕДСТАВЛЕННЯ ДАНИХ

Питання 5.2.

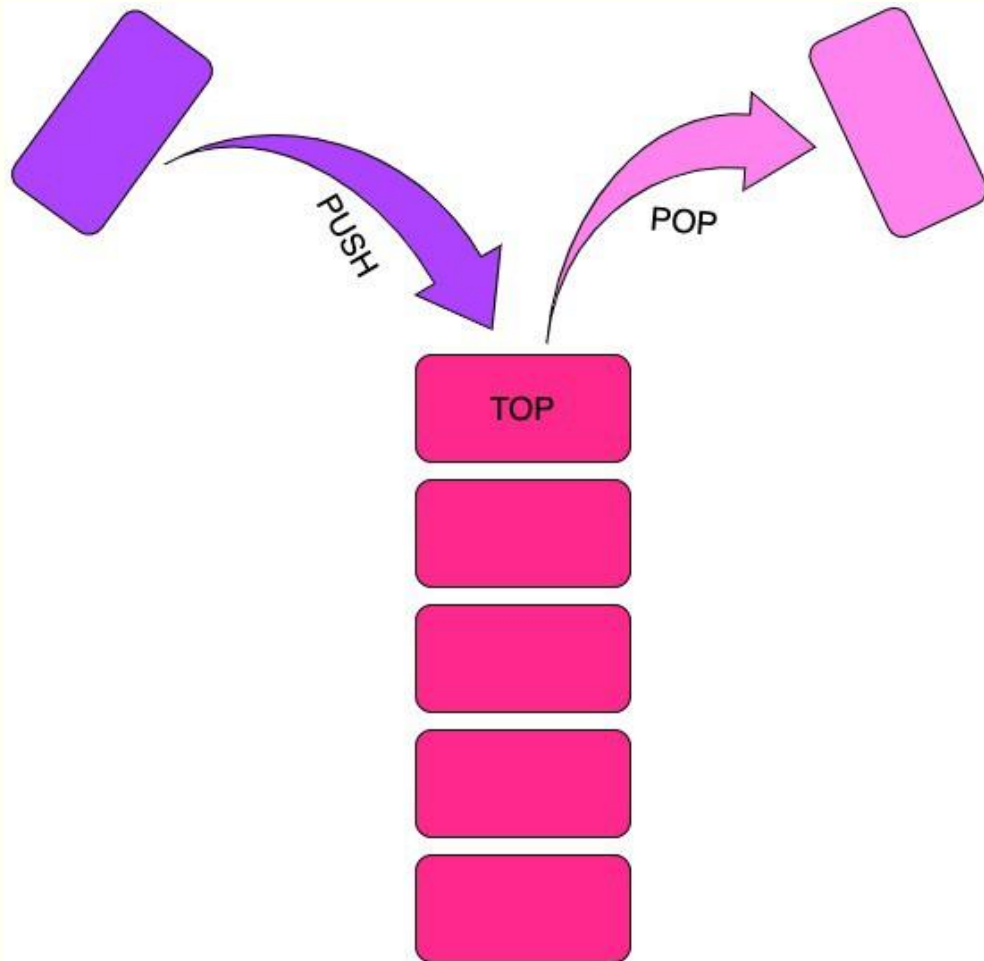
Поняття динамічних множин

- У той час, як математичні множини залишаються незмінними, множини, що обробляються під час роботи алгоритмів, можуть з часом розростатись, зменшуватись або підлягати іншим змінам.
 - Назвемо такі множини *динамічними*.
- Далі розглядатимемо деякі основні методи представлення скінченних динамічних множин та роботи з ними на ПК.
 - У багатьох алгоритмах використовується набір операцій: вставка елементів, їх видалення, перевірка елементу на приналежність множині.
 - Динамічна множина, що підтримує такі операції, називається *словником (dictionary)*.
- У типових реалізаціях динамічної множини кожен її елемент представляється деяким об'єктом.
 - Якщо наявний вказівник на об'єкт, то можна перевіряти та змінювати значення його атрибутів.
 - Можливо, що один з атрибутів об'єкта ідентифікує ключ (key). Якщо всі ключі різні, то динамічну множину можна представити у вигляді множини ключових значень.
 - Об'єкти можуть містити супутні дані (satellite data), які знаходяться в інших його атрибутах, проте не використовуються реалізацією множини.
 - У деяких множинах передбачається, що ключі належать повністю впорядкованій множині, що дозволяє визначити мінімальний елемент множини та говорити про найближчий елемент множини, що перевищує заданий.

Операції над динамічними множинами

- Ділять на дві категорії:
 - *Занути (queries)* – просто повертають інформацію про множину (Minimum(S), Maximum(S), Successor(S), Predecessor(S))
 - *Модифікуючі операції* (modifying operations – Insert(S, x), Delete(S, x))
- Стеки та черги – динамічні множини, елементи з яких видаляються за допомогою наперед заданої операції DELETE.
 - Першим зі стеку (stack) видаляється елемент, що поміщався в нього останнім – реалізується стратегія «останнім прийшов – першим вийшов» (LIFO).
 - У черзі завжди видаляється елемент, котрий міститься у множині довше за решту – реалізується стратегія «першим прийшов – першим вийшов».

Стеки та операції з ними



- Операція вставки Insert стосовно стеків називається **записом у стек (*push*)**, операція видалення Delete, що викликається без передачі аргументу, називається **зняттям зі стеку (*pop*)**.
 - Стек, що може вмістити не більше n елементів, можна реалізувати за допомогою масиву $S[1...n]$.
 - Останній поміщений в стек елемент має індекс $S.top$.
 - Якщо $S.top=0$, стек порожній.
 - Якщо відбувається спроба зняття з порожнього стеку, говорять, що він спустошується (*underflow*), що зазвичай призводить до помилки.
 - Якщо $S.top>n$, то стек вважається переповненим (*overflow*).

Операції зі стеком

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

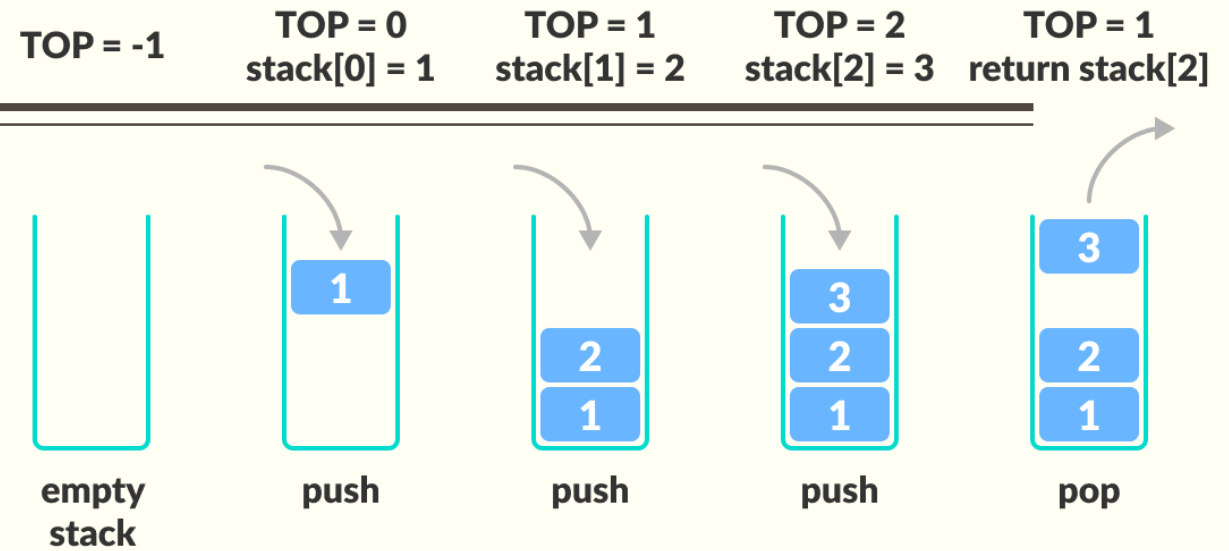
```
int MAXSIZE = 8;  
int stack[8];  
int top = -1;
```

```
int isempty() {  
    if(top == -1)  
        return 1;  
    else  
        return 0;  
}
```

```
int isfull() {  
    if(top == MAXSIZE)  
        return 1;  
    else  
        return 0;  
}
```

```
int peek() {  
    return stack[top];  
}
```

```
int pop() {  
    int data;  
  
    if(!isempty()) {  
        data = stack[top];  
        top = top - 1;  
        return data;  
    } else {  
        printf("Could not retrieve data, Stack is empty.\n");  
    }  
}  
  
int push(int data) {  
    if(!isfull()) {  
        top = top + 1;  
        stack[top] = data;  
    } else {  
        printf("Could not insert data, Stack is full.\n");  
    }  
}
```



Робота зі стеком

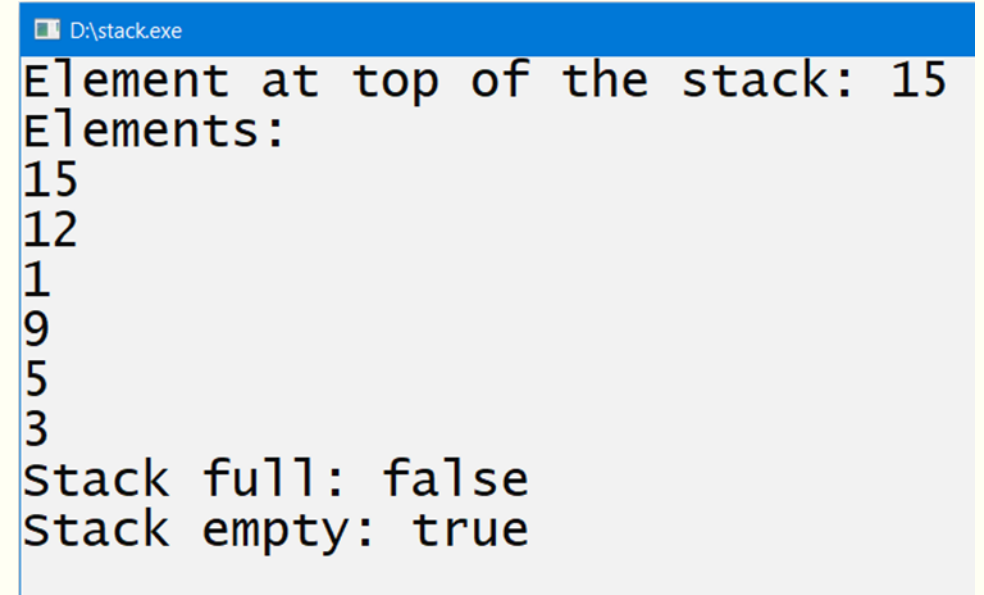
```
int main() {
    // push items on to the stack
    push(3);
    push(5);
    push(9);
    push(1);
    push(12);
    push(15);

    printf("Element at top of the stack: %d\n", peek());
    printf("Elements: \n");

    // print stack data
    while(!isempty()) {
        int data = pop();
        printf("%d\n", data);
    }

    printf("Stack full: %s\n", isfull()?"true":"false");
    printf("Stack empty: %s\n", isempty()?"true":"false");

    return 0;
}
```



```
D:\stack.exe
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true
```



stack.c

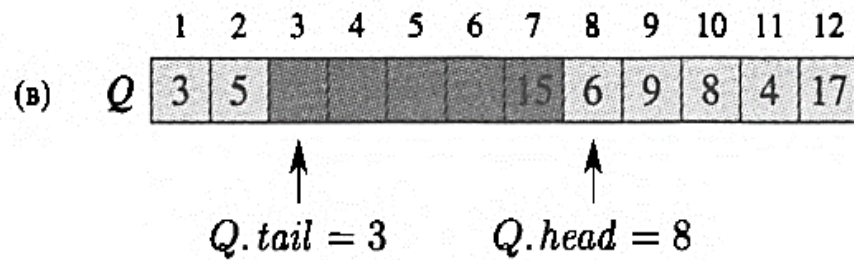
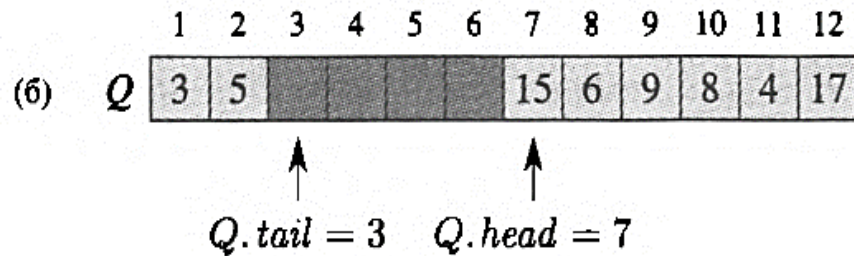
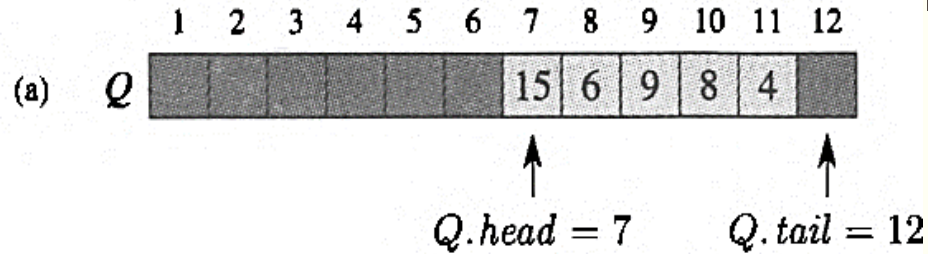
Застосування стеків

- Лежать в основі роботи виклику функцій та рекурсивних функцій.
 - Прямо зараз сотні функцій знаходяться в стеку викликів, що підтримується операційною системою в оперативній пам'яті.
 - При виклику функції спрацьовує резервування пам'яті (операція PUSH), а після повернення результату роботи функції виконується деаллокація пам'яті (операція POP).
- Функціональність «скасувати/відновити» (undo/redo) використовує стек.
 - Всі дії пушаються в стек, і як тільки натискається Ctrl+Z (скасування попередньої зміни), відбувається виштовхування цієї зміни зі стеку.
 - Кількість операцій скасування регулюється розміром стеку.
- Функціональність «останні», «нещодавні» (recently used).
 - Браузер використовує стек для відстеження нещодавно закритих вкладок та їх повторного відкриття.
- Редактор коду слідкує за тим, чи всі відкриваючі дужки були закриті.
 - Також за допомогою стеку вирази в коді (інфіксна форма запису) перетворюються в постфіксну форму запису (спочатку операнди, потім операція – тоді не потрібні дужки).

Застосування стеків

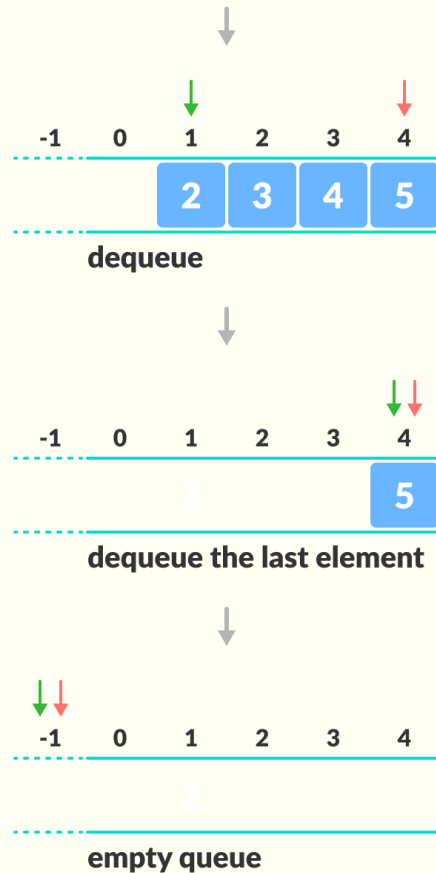
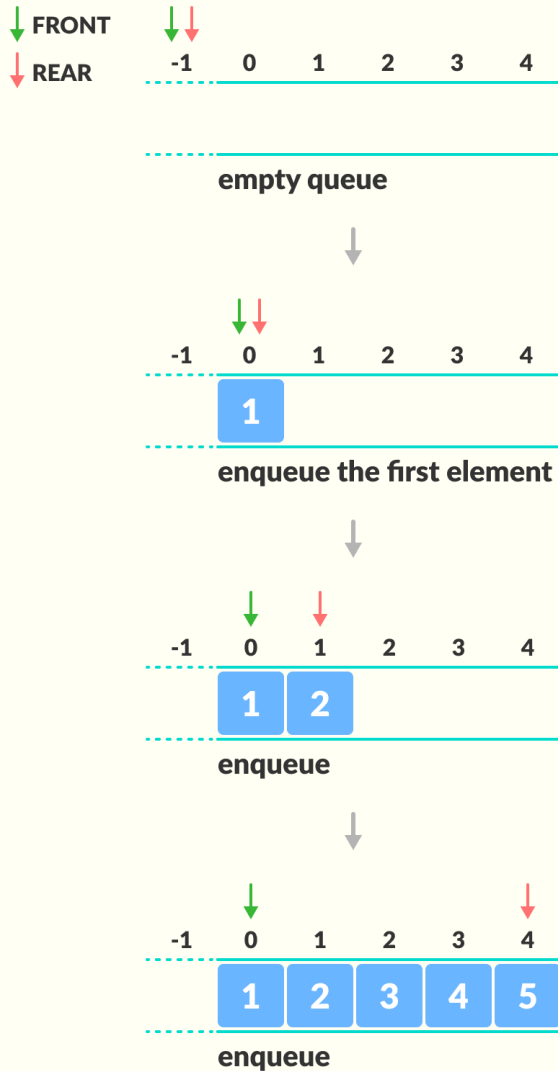
- Використовуються для реалізації алгоритмів пошуку з поверненням (backtracking).
 - Основна ціль алгоритму: якщо він обрав невірний шлях, слід «відкотитись» (back track) до попереднього стану.
 - Стани підтримуються за допомогою стеків.
 - На подібному принципі можна реалізувати алгоритм гри в «хрестики-нулики».
- Підвищують ефективність алгоритмів. Деякі алгоритми використовують стеки та їх властивості.
 - Наприклад, алгоритм Грехема для знаходження опуклої оболонки для скінченної множини точок.
 - Задача знаходження найближчого меншого чи більшого значення в масиві.

Черги та операції з ними



- Для черг операції вставки та видалення елементу називаються **Enqueue** та **Dequeue** відповідно.
 - Аналогічно до операції **pop**, **Dequeue** не має аргументів.
 - Черга подібна реальній черзі людей: у неї є **голова** (*head, front*) та **хвіст** (*tail, rear*).
 - Коли елемент поміщається в чергу, то займає місце в хвості. Виводиться елемент з голови черги.
 - За допомогою масиву на n елементів можна реалізувати чергу на $n-1$ елемент.
- Елементи черги розташовуються в комірках $Q.head, Q.head + 1, \dots, Q.tail - 1$.
 - Якщо $Q.head = Q.tail$, черга порожня.
 - На початку виконується умова $Q.head = Q.tail = 1$.
 - Якщо $Q.head = Q.tail + 1$ або $Q.head = 1$ і $Q.tail = Q.length$, черга переповнюється при додаванні елементу.

Операції з нециклічними чергами



```
#include <stdio.h>
#define SIZE 5
```

```
int items[SIZE], front = -1, rear = -1;
```

```
void enQueue(int value) {
    if (rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else {
        if (front == -1)
            front = 0;
        rear++;
        items[rear] = value;
        printf("\nInserted -> %d", value);
    }
}
```

```
void deQueue() {
    if (front == -1)
        printf("\nQueue is Empty!!");
    else {
        printf("\nDeleted : %d", items[front]);
        front++;
        if (front > rear)
            front = rear = -1;
    }
}
```

Операції з нециклічними чергами

```
int main() {
    //deQueue is not possible on empty queue
    deQueue();

    //enQueue 5 elements
    enQueue(1);
    enQueue(2);
    enQueue(3);
    enQueue(4);
    enQueue(5);

    //6th element can't be added to queue because queue is full
    enQueue(6);

    display();

    //deQueue removes element entered first i.e. 1
    deQueue();

    //Now we have just 4 elements
    display();

    return 0;
}
```

```
// Function to print the queue
void display() {
    if (rear == -1)
        printf("\nQueue is Empty!!!");
    else {
        int i;
        printf("\nQueue elements are:\n");
        for (i = front; i <= rear; i++)
            printf("%d ", items[i]);
    }
    printf("\n");
}
```

D:\noncircular_queue.exe

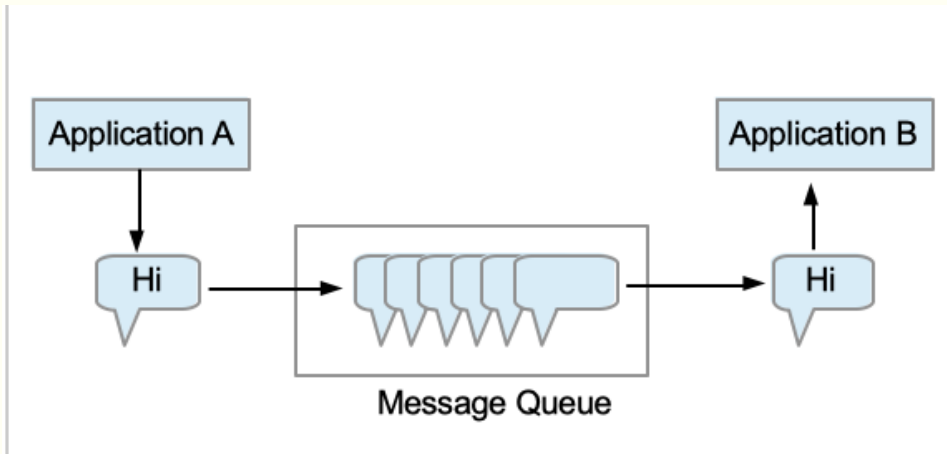
```
Queue is Empty!!
Inserted -> 1
Inserted -> 2
Inserted -> 3
Inserted -> 4
Inserted -> 5
Queue is Full!!
Queue elements are:
1 2 3 4 5

Deleted : 1
Queue elements are:
2 3 4 5
```



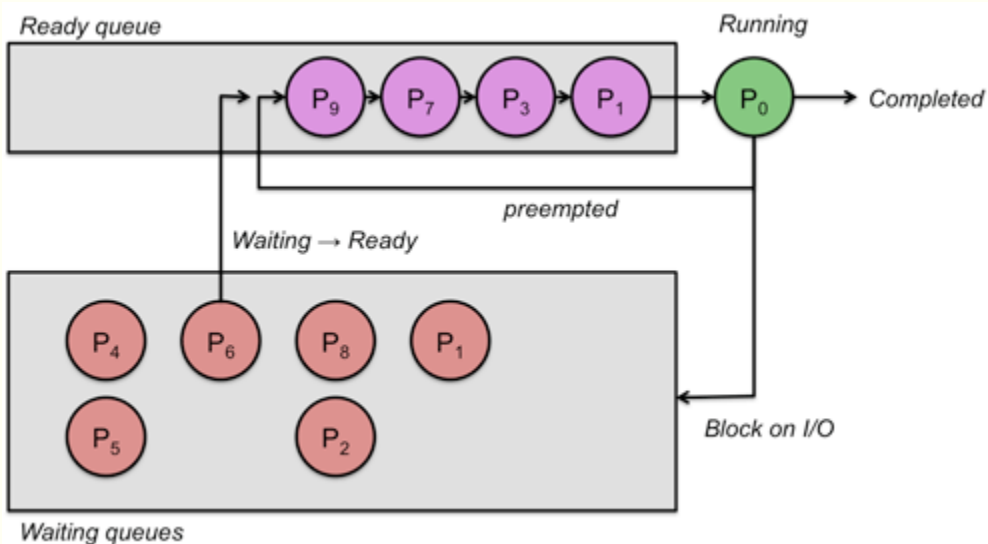
noncircular_queue.c

Застосування черг



■ **Черги повідомлень (*Message Queues*)** – загальна концепція взаємодії між процесами в ОС.

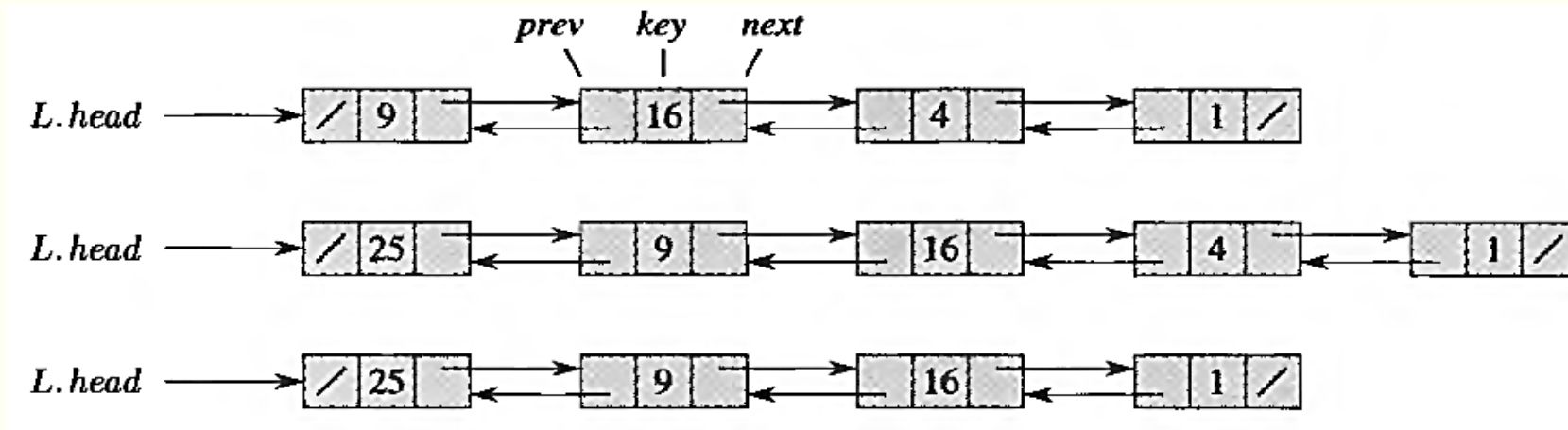
- Процес-відправник надсилає повідомлення (дані), проте процес-приймач не може отримати їх негайно, оскільки не працює чи зайнятий.
- Тоді повідомлення ставиться в чергу, і коли приймач буде готовий, воно передається йому та видаляється з черги.



■ **Планування виконання процесів (*Process Scheduling*)**.

- Усі процеси, що працюють в ОС зараз, спочатку чекають у черзі готовності (ready queue), очікуючи передачі їм у розпорядженням процесорних потужностей.
- Різні алгоритми планування по-різному вирішують, який процес обирати наступним для виконання відповідно до різних критеріїв, на зразок пріоритетності.

Зв'язні списки та операції з ними



- **Зв'язний список (*linked list*)** – структура даних, у якій елементи розташовані в лінійному порядку.
 - На відміну від масивів, порядок визначається не індексами, а вказівниками на кожен об'єкт.
 - Якщо $x.\text{prev}=\text{NIL}$, елемент x списку L називається головним, а якщо $x.\text{next}=\text{NIL}$, елемент x списку L називається хвостовим.
- Списки можуть бути різних видів:
 - **Одно- та двозв'язними.** Якщо список однозв'язний, відсутній вказівник на попередній елемент.
 - **Відсортованими або ні.** Якщо список відсортовано, то його лінійний порядок відповідає лінійному порядку його ключів.
 - **Кільцевими чи не кільцевими.** Якщо список кільцевий (*circular*), то вказівник *prev* його головного елемента вказує на його хвіст, а вказівник *next* хвостового елемента – на головний елемент.

Операції з двозв'язними списками

■ Виділяємо пам'ять

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
    int data;
    int key;

    struct node *next;
    struct node *prev;
};

//this link always point to first Link
struct node *head = NULL;

//this link always point to last Link
struct node *last = NULL;

struct node *current = NULL;
```

```
//is list empty
bool isEmpty() {
    return head == NULL;
}

int length() {
    int length = 0;
    struct node *current;

    for(current = head; current != NULL; current = current->next){
        length++;
    }

    return length;
}
```

Операції з двозв'язними списками

LIST-SEARCH(L, k)

```
1  $x = L.head$   
2 while  $x \neq \text{NIL}$  и  $x.key \neq k$   
3    $x = x.next$   
4 return  $x$ 
```

LIST-INSERT(L, x)

```
1  $x.next = L.head$   
2 if  $L.head \neq \text{NIL}$   
3    $L.head.prev = x$   
4  $L.head = x$   
5  $x.prev = \text{NIL}$ 
```

LIST-DELETE(L, x)

```
1 if  $x.prev \neq \text{NIL}$   
2    $x.prev.next = x.next$   
3 else  $L.head = x.next$   
4 if  $x.next \neq \text{NIL}$   
5    $x.next.prev = x.prev$ 
```

- **Пошук у зв'язному списку.** Процедура List-Search(L, k) знаходить у списку L перший елемент з ключем k шляхом простого лінійного пошуку та повертає вказівник на знайдений елемент. Якщо елемент з ключем k відсутній у списку, повертається NIL.
- **Вставка у зв'язний список.** Якщо маємо елемент x , атрибут key якого попередньо встановлено, то процедура List-Insert вставляє елемент x на початок списку.
- **Видалення зі зв'язного списку.** У процедуру List-Delete необхідно передати вказівник на елемент x , після чого відбувається видалення x шляхом заміни вказівників.
 - Щоб видалити елемент із заданим ключем, спочатку потрібно знайти вказівник на нього (List-Search).

Операції з двозв'язними списками

■ Видалення елементів:

```
//delete first item
struct node* deleteFirst() {

    //save reference to first link
    struct node *tempLink = head;

    //if only one link
    if(head->next == NULL){
        last = NULL;
    } else {
        head->next->prev = NULL;
    }

    head = head->next;
    //return the deleted link
    return tempLink;
}
```

```
//delete link at the last location
struct node* deleteLast() {
    //save reference to last link
    struct node *tempLink = last;

    //if only one link
    if(head->next == NULL) {
        head = NULL;
    } else {
        last->prev->next = NULL;
    }

    last = last->prev;

    //return the deleted link
    return tempLink;
}
```

```
//delete a link with given key

struct node* delete(int key) {

    //start from the first link
    struct node* current = head;
    struct node* previous = NULL;

    //if list is empty
    if(head == NULL) {
        return NULL;
    }

    //navigate through list
    while(current->key != key) {
        //if it is last node

        if(current->next == NULL) {
            return NULL;
        } else {
            //store reference to current link
            previous = current;

            //move to next link
            current = current->next;
        }
    }
}
```

```
//found a match, update the link
if(current == head) {
    //change first to point to next link
    head = head->next;
} else {
    //bypass the current link
    current->prev->next = current->next;
}

if(current == last) {
    //change last to point to prev link
    last = current->prev;
} else {
    current->next->prev = current->prev;
}

return current;
}
```


Операції з двозв'язними списками

■ Вставка елементів:

```
//insert link at the first location
void insertFirst(int key, int data) {

    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    } else {
        //update first prev link
        head->prev = link;
    }

    //point it to old first link
    link->next = head;

    //point first to new first link
    head = link;
}
```

```
//insert link at the last location
void insertLast(int key, int data) {

    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    } else {
        //make link a new last link
        last->next = link;

        //mark old last node as prev of new link
        link->prev = last;
    }

    //point last to new last node
    last = link;
}
```

```
bool insertAfter(int key, int newKey, int data) {
    //start from the first link
    struct node *current = head;

    //if list is empty
    if(head == NULL) {
        return false;
    }

    //navigate through list
    while(current->key != key) {

        //if it is last node
        if(current->next == NULL) {
            return false;
        } else {
            //move to next link
            current = current->next;
        }
    }

    //create a link
    struct node *newLink = (struct node*) malloc(sizeof(struct node));
    newLink->key = newKey;
    newLink->data = data;

    if(current == last) {
        newLink->next = NULL;
        last = newLink;
    } else {
        newLink->next = current->next;
        current->next->prev = newLink;
    }

    newLink->prev = current;
    current->next = newLink;
    return true;
}
```

Операції з двозв'язними списками

- Відображаємо вміст списку:

```
//display the list in from first to last
void displayForward() {

    //start from the beginning
    struct node *ptr = head;

    //navigate till the end of the list
    printf("\n[ ");

    while(ptr != NULL) {
        printf("(%d,%d) ",ptr->key,ptr->data);
        ptr = ptr->next;
    }

    printf(" ]");
}
```

```
//display the list from last to first
void displayBackward() {

    //start from the last
    struct node *ptr = last;

    //navigate till the start of the list
    printf("\n[ ");

    while(ptr != NULL) {

        //print data
        printf("(%d,%d) ",ptr->key,ptr->data);

        //move to next item
        ptr = ptr ->prev;
    }
}
```

Операції з двозв'язними списками

■ Робота зі списком:

```
void main() {
    insertFirst(1,10);
    insertFirst(2,20);
    insertFirst(3,30);
    insertFirst(4,1);
    insertFirst(5,40);
    insertFirst(6,56);

    printf("\nList (First to Last): ");
    displayForward();

    printf("\n");
    printf("\nList (Last to first): ");
    displayBackward();

    printf("\nList , after deleting first record: ");
    deleteFirst();
    displayForward();

    printf("\nList , after deleting last record: ");
    deleteLast();
    displayForward();

    printf("\nList , insert after key(4) : ");
    insertAfter(4,7, 13);
    displayForward();

    printf("\nList , after delete key(4) : ");
    delete(4);
    displayForward();
}
```

D:\doubly_linked_list.exe

```
List (First to Last):
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

List (Last to first):
[ (1,10) (2,20) (3,30) (4,1) (5,40) (6,56)
```



doubly_linked_list.c



ДЯКУЮ ЗА УВАГУ!

Наступне питання: залік!