Data Structure

Lecture 6

Dr. Ahmed Fathalla

Exam

1. The postfix form of the expression (A+ B)*(C*D- E)*F / G is?
   a) AB+ CD*E – FG /
   b) AB + CD* E – F G /
   c) AB + CD* E – *F *G /
   d) AB + CDE * – * F *G /

2. Which of the following properties is associated with a queue?
   a) First In Last Out
   b) First In First Out
   c) Last In First Out
   d) Last In Last Out

3. In a circular queue, how do you increment the rear end of the queue?
   a) rear++
   b) (rear+1) % CAPACITY
   c) (rear % CAPACITY)+1
   d) rear–

4. What is the need for a circular queue?
   a) effective usage of memory
   b) easier computations
   c) to delete elements based on priority
   d) implement LIFO principle in queues

5. In linked-based implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?
   a) Only front pointer
   b) Only rear pointer
   c) Both front and rear pointer
   d) No pointer will be changed

6. In linked-based implementation of a queue, where does a new element be inserted?
   a) before head
   b) after head
   a) before tail
   b) after tail

Exam

7. What does the following **A** operation do?

```
int A(queue *q)
{
        if (!IsEmpty(q))
                return q->entry[q->front];
        else
                return -1
}
```

a) Dequeue
b) Enqueue
c) Return the front element
d) Return the last element

8. In linked-based implementation of a queue, from where is an item served?

a) At the head of link list
b) At the centre position in the link list
c) At the tail of the link list
d) Node before the tail

9. The optimal data structure used to solve Tower of Hanoi is __________

a) Tree
b) Heap
c) Priority queue
d) Stack

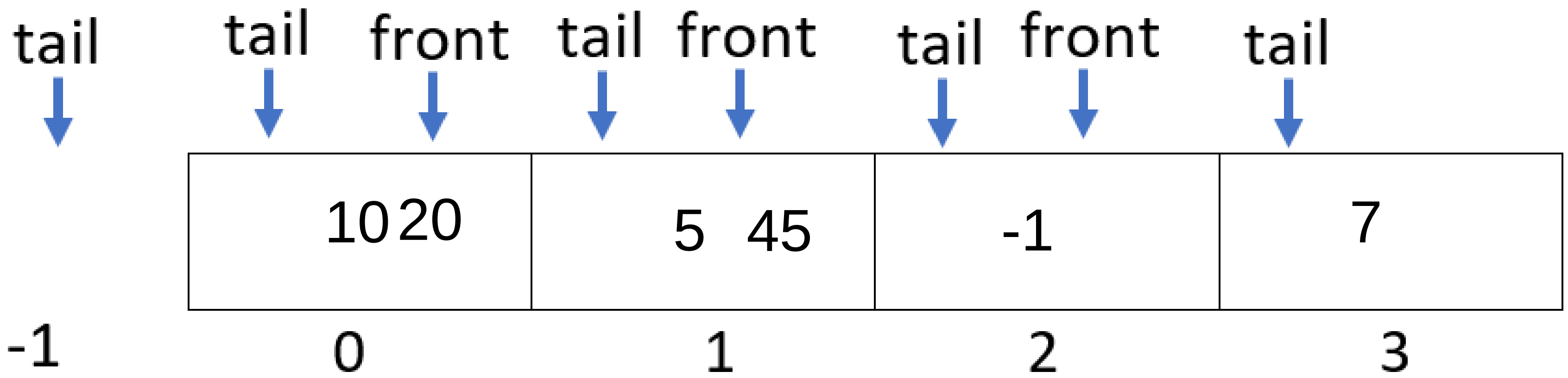
10. What is the number of moves required to solve Tower of Hanoi problem for k disks?

a) $2*k – 1$
b) $2*k + 1$
c) $2^k + 1$
d) $2^k – 1$

Exam

Question (2): Draw the queues after each operation (e.g., enqueue and dequeue), using an array-based circular of size 4. Consider an instance q of a Queue structure is created, then what will be the queue state (entrylist, front, tail) after each operation. **[5 Marks]**

1. CreateQueue(&q)
2. Enqueue (10, &q)
3. Enqueue (5, &q)
4. Enqueue (-1, &q)
5. Enqueue (7, &q)
6. Dequeue(&qentery e,&q)
7. Dequeue(&qentery e,&q)
8. Enqueue (20, &q)
9. Enqueue(45, &q)



1. Overflow and Underflow Conditions

- A stack/queue may have a limited space depending on the implementation. We must implement check conditions to see if we are not adding or deleting elements more than it can maximum support.
- If the array is full and no new element can be accommodated, then the stack **Overflow condition** occurs.
  - For example, If top = MAXSIZE, then, The stack is in overflow condition.
- The **Underflow condition** checks if there exists any item before popping/dequeue from the stack/queue.
  - Similarly, If top = 0, then, The stack is in underflow condition.
- **Underflow** and **Overflow** conditions occurs when a stack/queue is implemented using arrays.

2. Pre- & Post-condition

- Task: recheck the online material and review the pre-condition and post-condition of different methods for Stack and Queue. [Link-to-the-online-material]
- For example:

```
/*Pre: The stack is initialized and not full
  Post: The element e has been stored at the top of
the stack; and e does not change*/
void Push(StackEntry e, Stack *ps){
     ps->entry[ps->top++]=e;
}
```

3. Analysis of algorithms

- Task: using the same online material in the previous slide.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

	Array-based implementation	Linked implementation
Pop	$\Theta(1)$	$\Theta(1)$
Push	$\Theta(1)$	$\Theta(1)$
CreateStack	$\Theta(1)$	$\Theta(1)$
StackSize	$\Theta(1)$	$\Theta(1)$
TraverseSack	$\Theta(N)$	$\Theta(N)$
ClearStack	$\Theta(1)$	$\Theta(N)$

Depends on the implementation

3. Analysis of algorithms (Sec. 1.10)

- **Big-O Notation (O-notation)**

  Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

- **Omega Notation (Ω-notation)**

  Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

- **Theta Notation (Θ-notation)**

  Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

General Lists

Motivation: Why Lists?

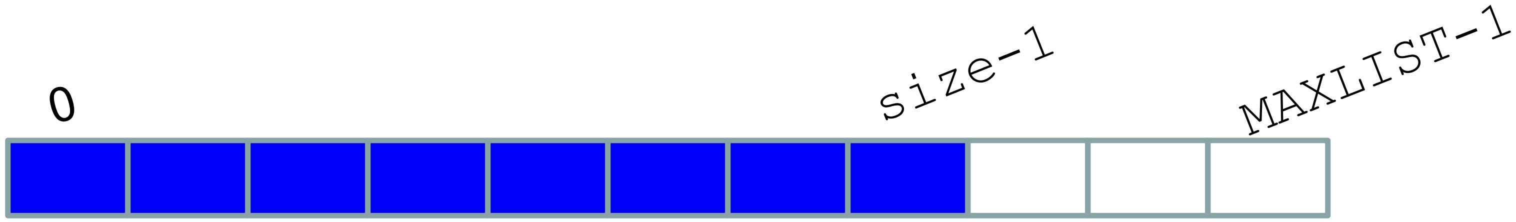
- **In a general list:**
  - new values are added in position determined by the user.
  - Element is removed from a position determined by the user.

- **Important notice:**
  - if we keep adding and removing from the first position (the head of the list) the general list will behave as a stack.

  - If we keep adding from one end and removing from another end the list will behave as a queue.

- **Application:**
  - In queues, sometimes we need a priority for some elements. We may need to put an emergency call prior to others.

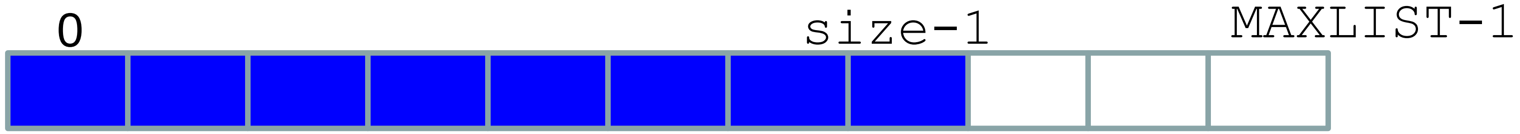
Motivation: How it works?



- This is a list (just a logical view, no implementation yet) with number of entries equals to `size`
- We can add a new element in position $0 \leq p \leq size$.
- However, we delete from $0 \leq p \leq size-1$.
- Now, let us be rigorous and define lists.

<u>Definition:</u> A ***general list*** of elements of type T is a finite sequence of elements of T together with the following operations:

1. **Create** the list, leaving it empty.
2. Determine whether the list is **empty** or not
3. Determine whether the list is **full** or not
4. Find the **size** of the list.
5. **Insert** a new entry in the position $0 \le p \le size$.
6. **Delete** an entry from the position $0 \le p \le size-1$
7. **Traverse** the list, visiting each entry
8. **RetrieveItem**
9. **ReplaceItem**
10. **Clear** the list to make it empty




```
void InsertList(int p, ListEntry e, List *pl);
```

Precondition:

1- The list pl has been created.

2- not full (**Overflow condition**)

3- 0 $\leq$p$\leq$ size.

Postcondition:

1- e has been inserted at position p

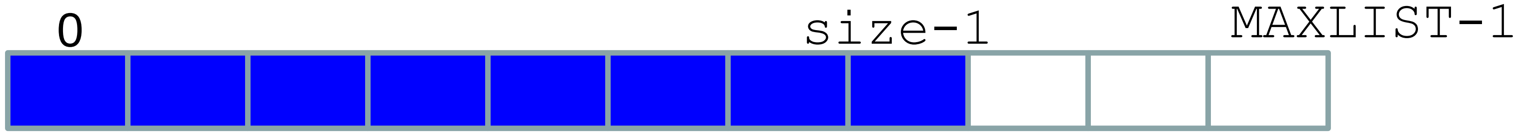
2- all elements at old positions p, p+1, …, size-1 are **incremented** by 1.

3- size increases by 1.

```
void DeleteList(int p, ListEntry *pe, List *pl);
```

Precondition: The list pl has been created, not empty (**Underflow condition**), and 0 $\leq$ p $\leq$ size-1.

Postcondition: e has been retrieved from position p, and all elements at old positions p+1, …, size-1 are **decremented** by 1. size decreases by 1.




```
void RetrieveItem (int p, ListEntry *pe, List *pl);
```

same precondition as DeleteList. And the list is unchanged

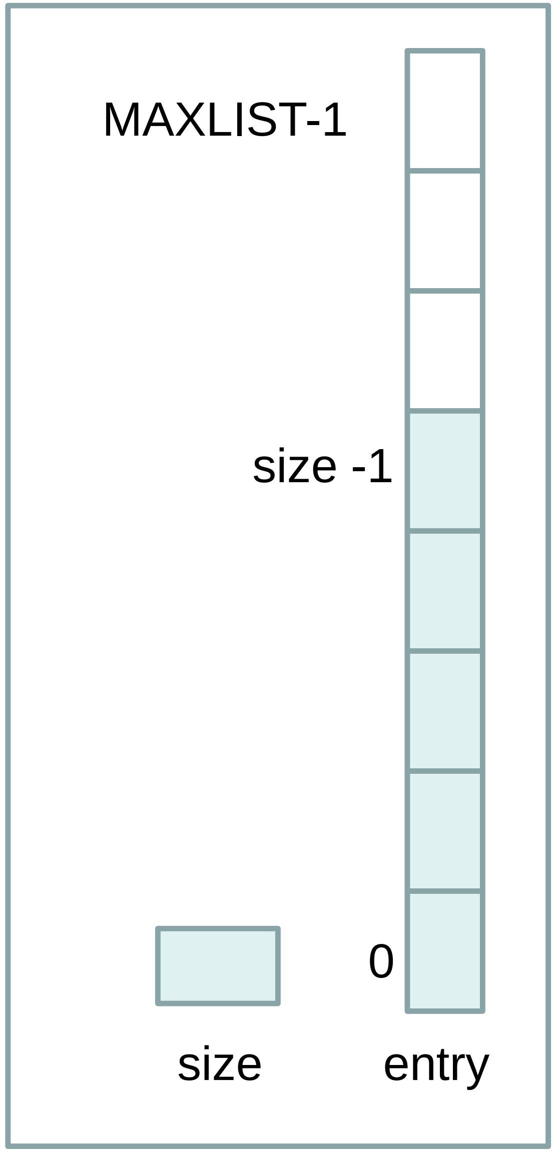
```
void ReplaceItem(int p, ListEntry e, List *pl);
```

same precondition. And the element is replaced

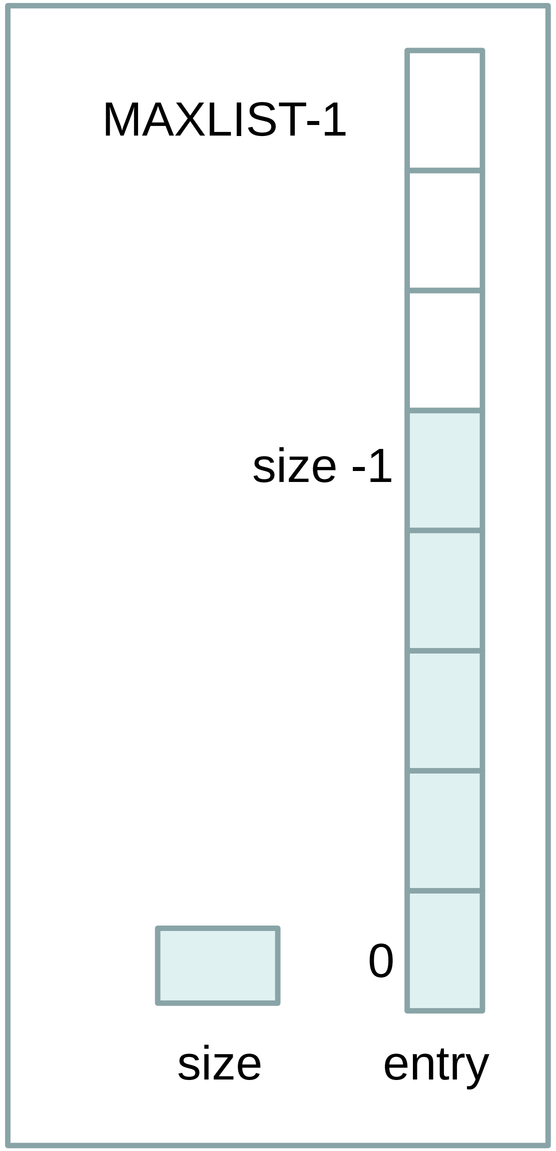
Other functions have similar pre- and post-conditions to the Queue and Stack.

Now let us start the Contiguous (array-based) Implementation

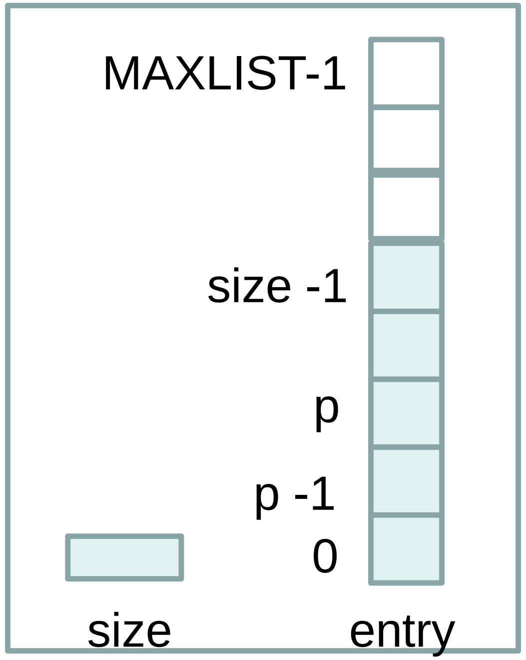
```c
/*List.h*/

struct List{
      ListEntry entry[MAXLIST];
      int size;
};

void  CreateList  (List *);
int   ListEmpty   (List *);
int   ListFull    (List *);
int   ListSize    (List *);
void  DestroyList (List *);
void  InsertList  (int, ListEntry, List *);
void  DeleteList  (int, ListEntry *, List *);
void  TraverseList(List *);
void  RetrieveItem(int, ListEntry *, List *);
void  ReplaceItem (int, ListEntry, List *);
```



```
void CreateList(List *pl){
	pl->size=0;
} // Ω(1)

int ListEmpty(List *pl){
	return !pl->size;
} // Ω(1)

int ListFull(List *pl){
	return pl->size==MAXLIST;
} // Ω(1)

int ListSize(List *pl){
	return pl->size;
} // Ω(1)

void DestroyList(List *pl){
	pl->size=0;
} // Ω(1)
```



```c
/*0 <= p <= size*/
void InsertList(int p, ListEntry e, List *pl){
     int i;
     /*The loop shifts up all the elements in the range [p,
size-1] to free the pth  location*/
     for(i=pl->size-1; i>=p; i--)
          pl->entry[i+1]=pl->entry[i];
     pl->entry[p]=e;
     pl->size++;
} // Ω(n)
```



Special Cases are all the combination of the following:

p = 0 or p = size

size = 0

All the cases will work

Inserting one element requires too many shifting!!

```
/*0<= p <= size-1 and List not empty*/
void DeleteList(int p, ListEntry *pe, List *pl){
    int i;
    *pe=pl->entry[p];
    /*The loop shifts down all the elements in
    the range [p+1, size-1] to free the pth
    location*/
    for(i=p+1; i<=pl->size-1; i++)
        pl->entry[i-1]=pl->entry[i];
    pl->size--;
} // Ω(n)
```

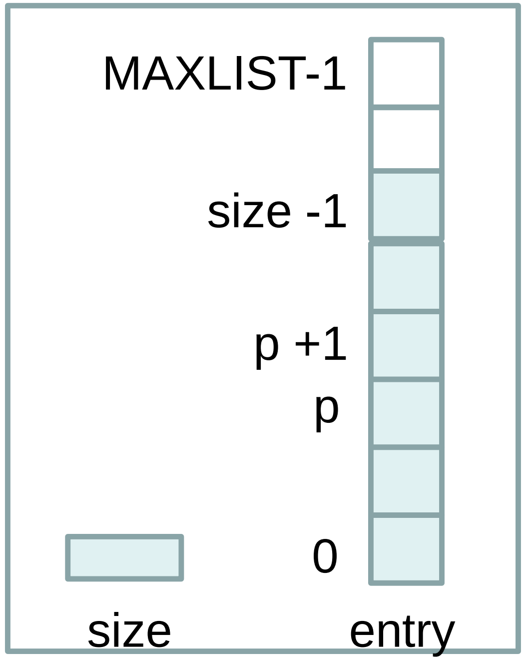
Special Cases are all the combination of the following:

p = 0 or p = size - 1

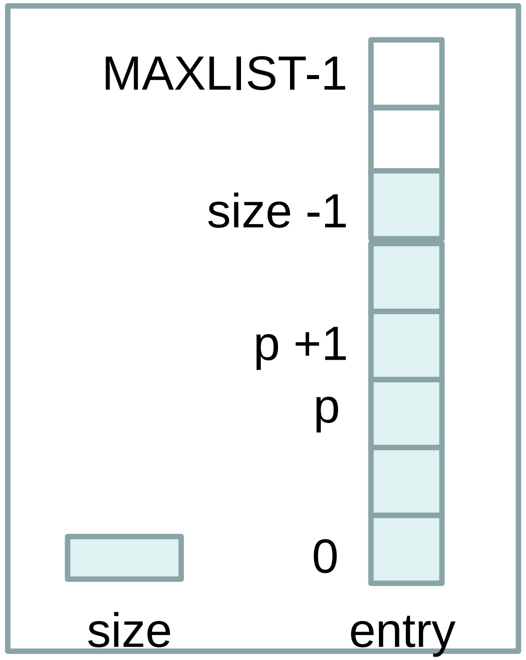
size = 1

All the cases will work

Deleting one element requires too many shifting!!



```
/* 0<= p <= size-1*/
void RetrieveItem(int p, ListEntry *pe, List *pl){
     *pe=pl->entry[p];
} // Ω(1)
/* 0<= p <= size-1*/
void ReplaceItem(int p, ListEntry e, List *pl){
     pl->entry[p]=e;
} // Ω(1)

void TraverseList(List* pl){
     int i;
     for(i=0; i<pl->size; i++)
          cout<<pl->entry[i];
} // Ω(n)
```



Issues at the user level:

How to insert at the beginning of the List?

```
InsertList(0, e, &l);
```

How to insert at the end of the List?

```
InsertList( ListSize(&l), e, &l);
```

Quiz

- How to use List as Stack.
- How to use List as Queue.
- How to do Replace method at the user-level.