4.2 Stack

4.2.1 Stack ADT

4.2.2 Array-based stack

4.2.3 Linked stack

72

Stack(栈）

LIFO List: Last In, First Out. 后进先出

Restricted form of list 有限制的线性表:

Insert and remove only at front of list.插入和删除只能发生在线性表的前端。

Notation:

- **Insert: Push**
- **Remove: Pop**
- **The accessible element(**前端元素**) is called TOP(**顶部元素**).**

73

Stack

The input of a stack is **abcde**, the output of the stack can't be ( ).

(A) edacb

(B) bcdae

(C) bcade

(D) aedcb

74

4.2.1 Stack ADT 栈的抽象数据类型定义

ADT Stack {

数据对象：

$D=\{ a_i \mid a_i \in ElemSet, i=0,2,...,n-1,\ n\geq 0 \}$

数据关系：

$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in ElemSet, i=1,...,n-1 \}$

约定a_{n-1}端为栈顶，a_0端为栈底。

基本操作：

} ADT Stack

75

栈的基本操作：

push( )

pop( )

topValue( )

Length( )

09:05

76

76

Stack ADT class

```
// Stack abtract class
template <class Elem> class Stack {
public:
  Stack() {}
  virtual ~Stack()  {}
  // Reinitialize the stack
  virtual void clear() = 0;
  // Push an element onto the top of the stack.
  virtual void push(const Elem&) = 0;
  // Remove the element at the top of the stack.
  virtual Elem pop() = 0;
  // Get a copy of the top element in the stack
  virtual Elem topValue() const = 0;
  // Return the number of elements in the stack.
  virtual int length() const = 0;
};
```

09:05

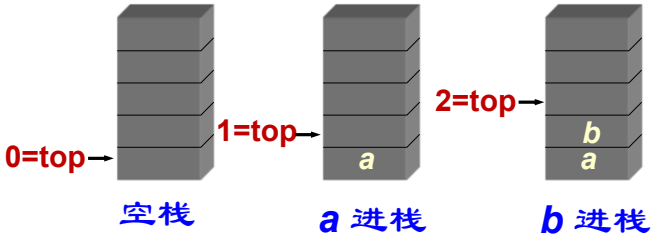
77

77

4.2.2 Array-based stack (顺序存储栈)

1个数组，2个整型变量即可描述stack

- ✓ 1个数组ListArray存储栈元素
- ✓ 1个整型变量maxSize描述数组的大小
- ✓ 1个整型变量top描述栈顶索引（同时也代表栈中元素个数）



maxSize=5

09:05

78

78

顺序栈的进栈和出栈原则

- 进栈时：将新元素放入top 位置，即ListArray [top]。

  再 top++
- 出栈时：先top--，

  再将top位置的元素,即ListArray [top]弹出。
- 栈满时再进栈将溢出出错;
- 栈空时再出栈将栈空出错。

09:05

79

79

Array-Based Stack class(1)

```
// Array-based stack implementation
template <class Elem>
class AStack{
private:
  int maxSize;      // Maximum size of stack
  int top;          // Index for top element
  Elem *listArray; // Array holding elements
public:
  AStack(int size) {
    top = 0;  maxSize = size;
    listArray = new Elem[maxSize];
  }
  ~AStack() { delete [] listArray; }
  int length() const { return top; }
  void clear()  { top = 0;  }
```

09:05 80

80

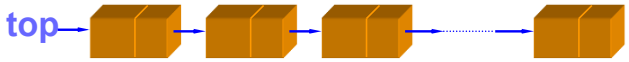
Array-Based Stack Class (2)

```
void push(const Elem& it) {
   Assert( top < maxSize, "Stack is full");
   listArray[top]= it;   top++;
 }
 Elem pop( ) {
   Assert( top > 0, "Stack is empty");
   top--;   return listArray[top];
 }
 Elem topValue( ) const {
    Assert( top > 0, "Stack is empty");
    return listArray[top-1];
 }
};
```

09:05 81

81

4.2.3 link-based stack (链式栈)



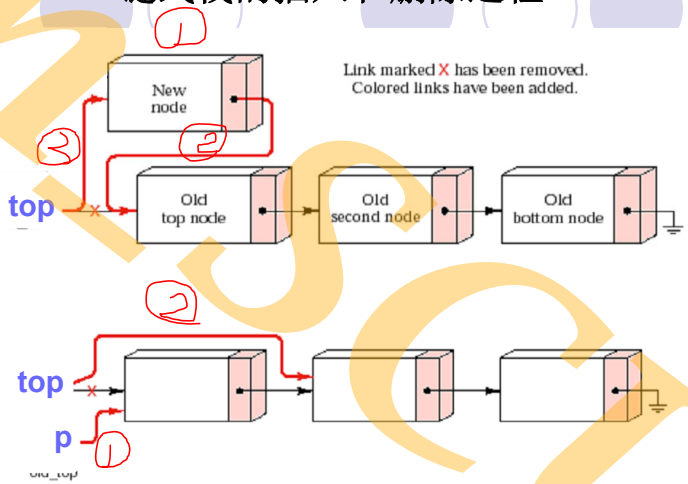
1个指针和1个整型变量即可描述一个链式栈
- ✓ 指向栈头结点的指针top
- ✓ 描述栈中元素个数的变量size

- 链式栈的栈顶在链头
- 链式栈无栈满问题，空间可扩充
- 有栈空问题

09:05 82

82

链式栈的插入和删除过程



09:05 83

83

Linked Stack class(1)

```
// Linked stack implementation
template <class Elem>
class LStack {
private:
  Link<Elem>* top; // Pointer to first elem
  int size;

public:
  LStack( )
  {  top = NULL;    size = 0; }
  ~LStack() { clear(); }  // Destructor
  void clear() {
      while(top != NULL) {
          Link<Elem>* temp = top;
          top = top->next;
          delete temp;  }
      size=0;
  }
```

09:05 84

84

Linked Stack Class (2)

```
  void push(Elem& it){
    top = new Link<Elem>(it,top);
    size++;
  }
  Elem pop(){
   Assert( top!=NULL, "Stack is empty");
   link<Elem> *temp = top;
   top = top->next;   size--;
   Elem  it = temp->element;
   delete temp;
   return it;
 }
  Elem topValue() const {
    Assert( top!=NULL, "Stack is empty");
    return top->element;
  }
  int length() const { return size;}
};
```

09:05 85

85

请思考

- What is the cost of the operations in arrayed-based stack and Linked stack?
- How about do space requirements of the array-based stack and Linked stack implementation?
- 自由链表可与链式栈结合以降低new， delete操作的时间吗？

09:05 86

86

栈的应用举例1：

十进制数N和其他d进制数的转换

除d求余法： 迭代 N = (N / d).....N % d

例如： $(1348)_{10} = (2504)_8$，其运算过程如下：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

09:05 87

87

伪代码：

```
void conversion ()
{
   InitStack( );          // 构造空栈
   cin >> N;              // 输入一个十进制数
   while(N)       {
      Push(N % 8);     // "余数"入栈
      N = N / 8;              // 非零"商"继续运算
   }
    while (!StackEmpty)     {
      e=Pop( ); cout << e;
   }
}
```

09:05 88

88

Main函数

```
……
#include  “Astack.h”
// #include  “Lstack.h”
void main()  {
  AStack<int>  myStack(100);
// LStack<int>  myStack;
  int N,M,d, e;
  cout<<"请输入一个十进制数：";    cin >>N;    M=N;
  cout<<"请输入拟转化的进制基数：";    cin >> d;
  while(N) {
          myStack.push(N%d);
          N=N/d;
  }
  cout<<"十进制数"<<M<<"  对应的”<<d<<“进制为：";
  while ( myStack.length() ) {
          e=myStack.pop();
          cout<<e;
  }
}
```

09:05 89

89

AStack.h (课件p80-81)

```
// Array-based stack implementation
template <class Elem>
class Astack  {
private:
    int maxSize;     // Maximum size of stack
    int  top;      // Index for top element
    Elem *listArray; // Array holding elements

public:
    AStack(int size) {
          top = 0;  maxSize = size;
           listArray = new Elem[maxSize];
    }

   ……
};
```

09:05 90

90

LStack.h (课件p29,p84-85)

```
// Singly-linked list node
template <class Elem> class Link {
public:
  Elem element; // Value for this node
  Link *next;   // Pointer to next node
  Link(const Elem& elemval,
       Link* nextval = NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval = NULL)
    { next = nextval; }
};

// Linked stack implementation
template <class Elem>   class LStack {
private:
   Link<Elem>* top; // Pointer to first elem
   int size;
public:
    ……
    ……
};
```

09:05 91

91

4.3 Queue

4.3.1 queue ADT

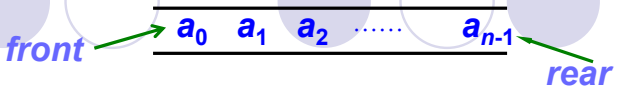
4.3.2 linked queue

4.3.3 array-based queue

09:05 92

92

Queues(队列)



FIFO List: First in, First Out

Restricted form of list:

Insert at one end（Rear）

remove from the other end（Front）

Notation:

- Insert: Enqueue（入队）
- Remove: Dequeue（出队）
- First element: Front (队头)
- Last element: Rear (队尾)

09:05 93

93

4.3.1 queue ADT

ADT Queue {

数据对象：

D = $\{a_i \mid a_i \in \text{ElemSet}, i=0,1,...,n-1, n\geq 0\}$

数据关系：

R1 = $\{ <a_{i-1},a_i> \mid a_{i-1}, a_i \in D, i=1,...,n-1\}$

基本操作：

} ADT Queue

09:05 94

94

队列的基本操作：

EnQueue( )

DeQueue( )

HeadValue( )

Length( )

09:05 95

95

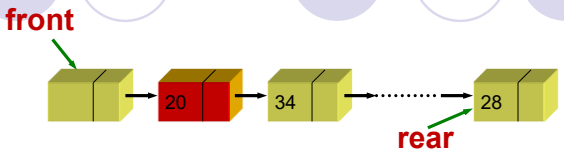
4.3.1 Queue ADT class

```
// Queue abtract class
template <class Elem> class Queue {
public:
  Queue()  {}
  virtual ~Queue()  {}
  // Reinitialize the queue
  virtual void clear() = 0;
  // Append an element into the rear of the queue.
  virtual void enQueue (const Elem&) = 0;
  // Remove the element at the front of the queue.
  virtual Elem deQueue() = 0;
  // get a copy of the front element in the queue
  virtual Elem frontValue() const = 0;
  // Return the number of elements in the queue.
  virtual int length() const = 0;
};
```

09:05 96

96

4.3.2 Linked queue（链式队列）



- front在链头，rear在链尾。跟Llist一样，有一专门的空数据结点作为头结点。
- 链式队列在进队时无队满问题，但在出队时有队空问题。
- 队空条件为 front->next == NULL

97

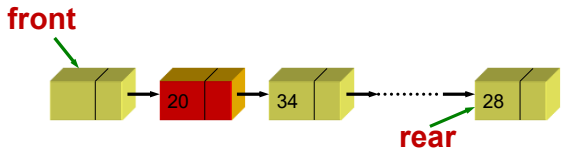
09:05

97

4.3.2 Linked queue（链式队列）

2个指针和1个整型变量即可描述一个链式栈

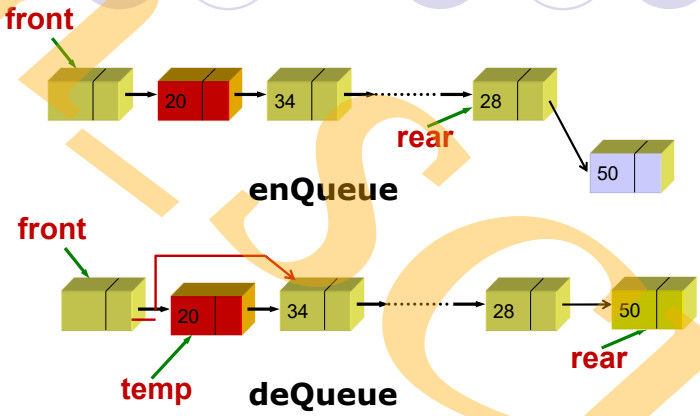
- ✓ 指向队头空数据结点的指针 front
- ✓ 指向队尾结点的指针 rear
- ✓ 描述队列中元素个数的变量size



09:05 98

98

链式队列的入队和出队过程



出队前需判断是否为空 99

09:05

99

Linked Queue class(1)

```
// Linked Queue implementation
template <class Elem>
class LQueue {
private:
  Link<Elem>* front; // Pointer to front elem
  Link<Elem>* rear; // Pointer to rear elem
  int size;       // Count number of elems
public:
  LQueue() {   front = rear = new Link<Elem>(); size = 0; }
  ~LQueue() { clear(); delete front; }
  void clear() {
     while(front->next!= NULL) {
          rear = front;   front = front->next;
          delete rear;
      }
      rear = front;  size=0;
  }
```

09:05 100

100

Linked Queue Class (2)

```
void enQueue(Elem& it) {
    rear->next =  new Link<Elem>(it,NULL);
     rear=rear->next;  size++;
  }
  Elem deQueue( ){
       Assert (  size!=0,  " Queue is empty"); ;
       Link<Elem> *temp = front->next;
       it = temp->element;     front->next = temp->next;
       if (rear == temp)   rear = front;
       size--; delete temp;
       return it;
   }
```

09:05 101

101

Linked Queue Class (3)

```
 Elem frontValue( ) const {
     Assert (  size!=0,  " Queue is empty") ;
     return   front->next->element;
   }
   int length()  const { return size;    }
};
```

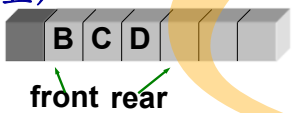
09:05 102

102

4.3.3 array-based queue(顺序存储队列)

1个数组，3个整型变量可描述顺序队列

- ✓ 用1个数组存储队列元素
- ✓ 1个整型变量maxSize描述数组的大小
- ✓ 1个整型变量front描述队头索引（指向队头的当前位置）
- ✓ 1个整形变量rear描述队尾索引（一般指向队尾的后一个位置）

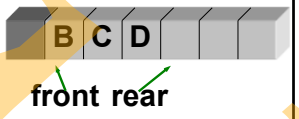


09:05 103

103

队列的进队和出队原则

- 进队时：先将新元素插入rear位置，然后rear=rear+1。
- 出队时：先将下标为 front 的元素取出，然后 front=front+1。



- 队初始化：front = rear = 0
- 队满时再进队将溢出出错；
- 队空时再出队将队空出错。

104

队列的进队和出队示例

105

伪溢出(pseudo-overflow)最直接的解决办法

每次出队都将剩余所有元素向前移动。即front永远为0

虽然可解决伪溢出问题，但出队操作复杂度由Θ(1) 增加到 Θ(n)

此解决办法不好！丢弃!!

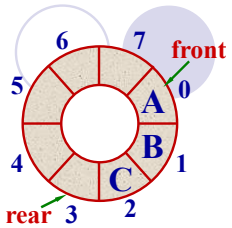
106

更好的解决办法——环形队列

将存放队列元素的数组首尾相接，形成循环(环形)队列 Circular Queue 。

107

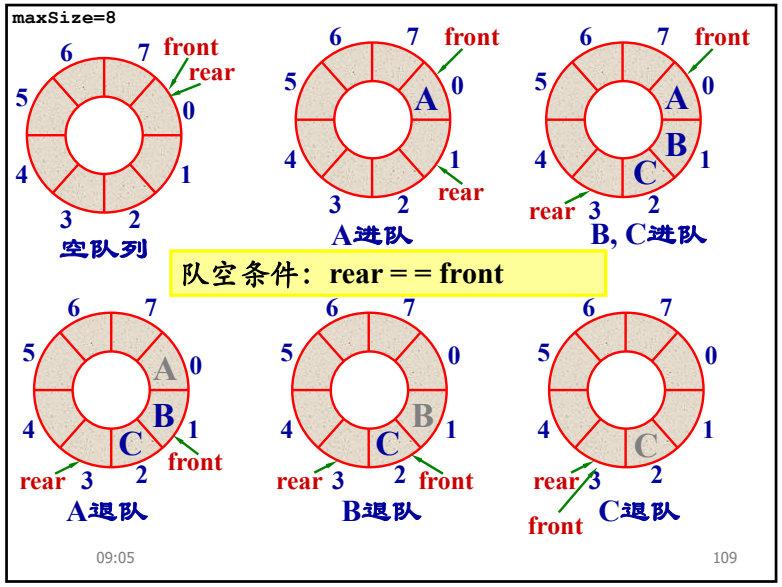
循环队列 (Circular Queue)



- 存放队列的数组被当作首尾相接的环。
- 队头、队尾索引加1后若等于数组最大尺寸maxSize，则从maxSize -1直接进到0，可用取模运算实现。
  - ☞队头指针加1: front = (front+1) % maxSize
  - ☞队尾指针加1: rear = (rear+1) % maxSize
  - ☞队列初始化（空对列）： front = rear = 0

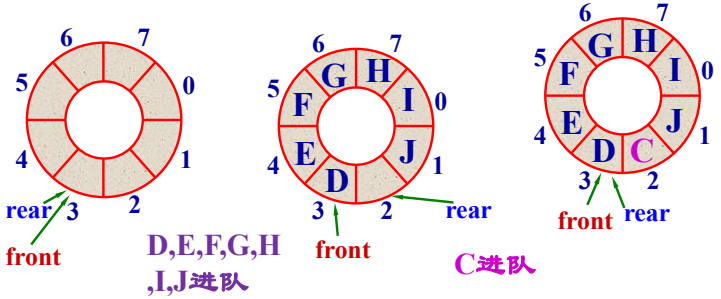
09:05 108

108



109

队空条件：rear = = front



队满条件：rear = = front

09:05 110

110

解决方案一：

设置计数器count，记录队列中的元素个数

- ☞队列初始化：front = rear = count = 0
- ☞队空条件：count = 0
- ☞队满条件：count = maxSize

1个数组(listArray)，4个整型变量(maxSize, front，rear，count)描述循环顺序队列

09:05 111

111

解决方案一总结：

- 将队列首尾相连构成循环队列（逻辑上）
  - 解决了伪溢出问题
  - 出队：front=(front+1)% maxSize
  - 入队：real=(real+1)% maxSize
- 设置计数器count
  - 解决了队满队空条件相同的问题
  - 队空： count=0
  - 队满： count=maxSize

09:05 112

112

Circular AQueue class(方案一)

```
// Array-based circular queue implementation one
template <class Elem>
class CAQueue {
private:
  int maxSize;     // Maximum size of queue
  int front;    // Index for front element
  int rear;     // Index for rear element
  int count;
  Elem *listArray; // Array holding elements
public:
  CAQueue(int size=DefaultListSize) {
    front = rear = count = 0;  maxSize = size;
    listArray = new Elem[maxSize];
  }
  ~CAQueue() { delete [] listArray; }
  void clear()  { front = rear = count = 0;  }
```

09:05 113

113

Circular AQueue class(方案一)

```
int length() const  { return count;}
Elem frontValue( ) const {
  Assert ( count > 0, " Queue is empty");
   return  listArray[front];
}
void enQueue(const Elem& it)  {
    Assert ( count < maxSize, " Queue is full");
    listArray[rear]= it;
    rear = (rear+1)% maxsize ;   count++;
  }
  Elem deQueue  ()  {
      Assert ( count > 0,     " Queue is empty");
      Elem it = listArray[front];
      front=(front+1)%maxSize ; count--;
      return it;
  }
};
```

09:05 114

114

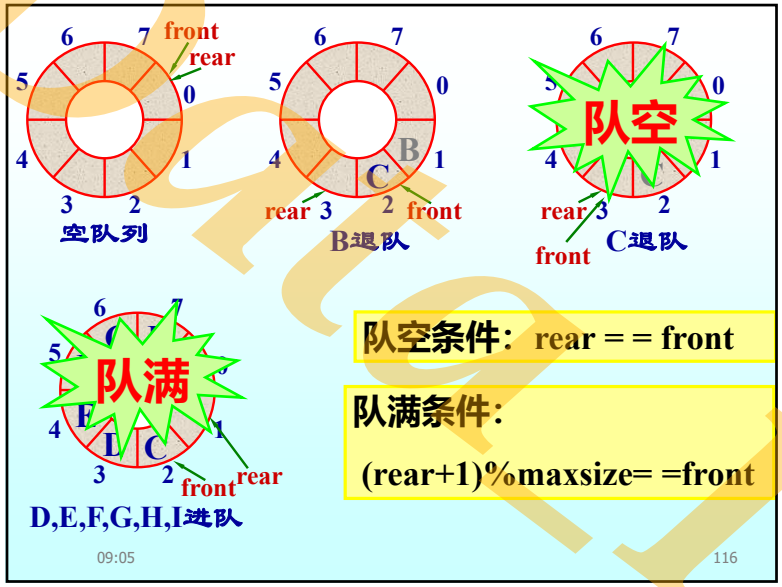
解决方案二：

在数组中始终留空一个单元不用。即rear所指的单元始终不用

- 队列初始化：front = rear = 0
- 队空条件：front = = rear
- 队满条件：(rear+1)% maxSize = = front
  - 当队尾再加1追上队头时，队列满
  - 当队列中元素个数达到maxSize-1时，队满

09:05 115

115



116

解决方案二总结：

- 将队列首尾相连构成循环队列（逻辑上）
  - 解决了伪溢出问题
  - 出队：front=(front+1)% maxSize
  - 入队：real=(real+1)% maxSize
- 留空一个单元不用
  - 解决了队满队空条件相同的问题
  - 队空： front = = rear
  - 队满： (rear+1)% maxSize = = front

09:05 117

117

Circular AQueue class(方案二)

```
// Array-based circular queue implementation two
template <class Elem>
class CAQueue {
private:
  int maxSize;      // Maximum size of queue
  int front;     // Index for front element
  int rear;      // Index for rear element
  Elem *listArray; // Array holding elements
public:
  CAQueue(int size) {
    front = rear = 0;  maxSize = size+1;
    listArray = new Elem[maxSize];
  }
  ~CAQueue() { delete [] listArray; }
  void clear() { front = rear = 0; }
  int length() const { return (rear+maxSize-front)%maxSize; }
```

09:05 118

118

Circular AQueue class(方案二)

```
void enQueue(const Elem& it) {
  Assert ( ((rear+1)%maxsize) ! =front, " Queue is full");
  listArray[rear]= it;
  rear = (rear+1)% maxSize;
}
Elem deQueue() {
  Assert ( length ( )! =0, " Queue is empty");
  Elem it = listArray[front];
  front=(front+1)%maxSize;
  return it;
}

Elem frontValue() const {
   Assert ( length ()! =0, " Queue is empty");
   return listArray[front];
}
};
```

09:05 119

119

请思考

- What is the cost of the operations in Circular Queue and Linked Queue implementation?
- How about do space requirements of the Circular Queue and Linked Queue implementation?
- Linked Queue 能与freelist 结合吗，若能，how?

09:05 120

120

An example of Stack and Queue

Given an non-empty queue Q, a empty stack S, a variable X, write an function to reverse the order of the elements in Q. （exercises 4.18）

```
template <class Elem>
void reverse(LQueue<Elem> & Q, LStack<Elem> & S)
{
   Elem X;
   while( Q.length ()) {
      X=Q.deQueue( );  S.push(X);  }
   while( S.length ()) {
      X=S.pop( );  Q.enQueue(X); }
}
```

09:05 121

121

Main.cpp

```
……
#include “LQueue.h”     // #include “CAQueue.h”
#include “LStack.h”     // #include “AStack.h”
template <class Elem>
void reverse(LQueue<Elem> & Q, LStack<Elem> & S)
{
   Elem X;
   ……
}
void main()  {
  LQueue<int> myQueue;   // CAQueue<int> myQueue(100);
  LStack<int> myStack;   // AStack<int> myStack(100);
  int i, A[10] = {1 2 3 4 5 6 7 8 9 10};
  for ( i=0; i<10; i++)    myQueue.enQueue(A[i]);
  reverse(myQueue, myStack);
}
```

09:05 122

122

LQueue.h (课件p29, p101-103)

```
// Singly-linked list node
template <class Elem> class Link {
public:
  Elem element; // Value for this node
  Link *next;   // Pointer to next node
  … …
};

// Linked Queue implementation
template <class Elem>
class LQueue {
private:
  Link<Elem>* front; // Pointer to front elem
  Link<Elem>* rear; // Pointer to rear elem
  int size;       // Count number of elems
public:
   ……
};
```

CAQueue.h (课件p114-1115或119-120)

09:05 123

123

这一章我们学到了

- List
  - List ADT
  - Array based list
  - Linked list
  - Freelists
  - Double linked list
- Stack----a special list
  - Stack ADT
  - Array based stack
  - Linked stack
- Queue---another special list
  - Queue ADT
  - Circular Aqueue---based on array
    - using a item count
    - always a space not use
  - LQueue

09:05 124

124

Chapter4 End

09:05 125

125