Lists, Stacks and Queues

Fall 2020

School of Software Engineering

South China University of Technology

Content

- List
- Stack
- Queue

List

What is List

- A list is a finite, ordered sequence of data items called elements.
  - Notation: $<a_0, a_1, \ldots, a_{n-1}>$
  - Each element has a position in the list.
  - Each element may be of arbitrary type, but all are of the same type
  - The length of a list is the number of elements currently stored
    - An empty list contains no elements
  - The beginning and the end of the list are, respectively, called the head and the tail
  - Common List operations are:
    - insert, append, delete/remove, find, isEmpty, prev, next, currPos, moveToPos, moveToStart, length, etc

List ADT

```
// List ADT
template <typename E> class List {
  public:
      List() {}
      virtual ~List() {}
      virtual bool insert(const E& item) = 0;
      virtual bool append(const E& item) = 0;
      virtual bool remove(E&) = 0;
      virtual void clear() = 0;
      virtual void moveToStart() = 0;
      virtual void moveToEnd() = 0;
      virtual void prev() = 0;
      virtual void next() = 0;
      virtual int currPos() const =0;
      virtual void moveToPos(int pos) = 0;
      ……
}
```

List Implementation

- Two standard list implementations
  - Array-based lists
  - Pointer-based lists (Linked lists)

List: Array Implementation

- Basic Idea:
  - Pre-allocate a big array of size **`MAX_SIZE`**
  - Keep track of current size using a variable **`count`**
  - Keep track of current position using a variable **`curr_position`**
  - Shift elements when you have to insert or remove

0	1	2	3	…	count-1		MAX_SIZE
A_1	A_2	A_3	A_4	…	A_N		

List: Array Implementation

insert Z in kth position

0	1	2	3	4	5			MAX_SIZE
A	B	C	D	E	F			

0	1	2	3	4	5	6		MAX_SIZE
A	B	Z	C	D	E	F		

Running time for N elements?

On average, must move half the elements to make room – assuming insertions at positions are equally likely

Worst case is insert at position 0. Must move all N items one position before the insert

This is O(N) running time.
Θ(1) for best case

Quickly?

List: Array Implementation

- insert the element at position curr
  - Shift left n-i-1 elements toward the tail

```
// insert element at the current position
void insert(const E& it) {
  Assert(listSize<maxSize, “Exceed capacity”);
  //shift Elements up
  for(int i=listSize; i>curr; i--)
      listArray[i] = listArray[i-1];
  listArray[curr] = it; // insert the element
  listSize++;  // increment list size
}
```

List: Array Implementation

- remove the element at position curr
  - Shift left n-i-1 elements toward the head

```
// Remove and return the current element
E remove() {
 Assert((curr>=0)&&(curr<listSize), "no element);
 E it = listArray[curr]; // Copy the element
 for(int i=curr; i<listSize-1; i++)
   // Shift them down
   listArray[i] = listArray[i+1];
   listSize--;    // Decrement size
   return it;
 }
```

Time cost – Θ(1) for best case;
Θ(n) for worst- and average cases

List: Array Implementation

- Other operations
  -

```
bool moveToPos(int pos) {
   Assert((pos>=0)&&(pos<listSize)), “out of range);
   curr = pos;
}

void moveToStart() { curr = 0; } //reset position
void moveToEnd() { curr = listSize; } //set at end

void prev() { if(curr!=0) curr--; }
void next() { if(curr < listSize) curr++; }

int Length() const  { return listSize; }
int currPos() const { return curr; }
```

Time cost – Θ(1) for best, worst- and average cases

List: Array Implementation

- Search for a value K in the list

```
//Return true if K is in list, otherwise,false
bool find(List<int>& L, int K) {
    int it;
    for(L.moveToStart(); L.currPos()<L.Length(); L.next())
{
            it = getValue();  //return value of curr element
            if (K == it) return true;  // Found it
    }
    return false;  // Not found
 }
```

Time cost – Θ(n) for worst- and average cases

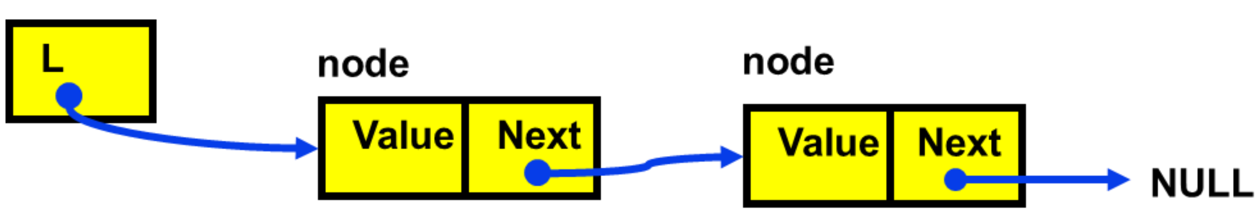
List: Array Implementation

- C++ STL – Vector
  - A growable array implementation of the List ADT
    - How to provide a grow-able array implementation of the List ADT

  - Textbook section 3.3 vector & Code vector.h and TestVector.cpp

List: Pointer Implementation

- Linked list
  - Use dynamic memory allocation which allocates memory for new list elements as needed
  - Elements are called nodes, which are linked using pointers.
    - Keep track of list by linking the nodes together
  - Change links when you want to insert or delete



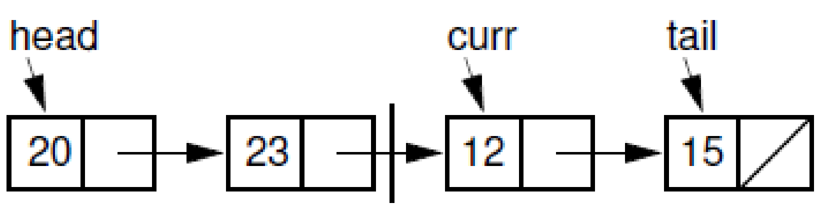
List: Pointer Implementation

```
// An implementation of a simple
// singly-linked list node

template <typename E> class Link {
   public:
   E element;  //value for this node
   Link *next;   //Pointer to next node in list
   //Constructors
   Link(const E& elemval, Link* nextval =NULL)
   { element = elemval;  next = nextval; }
   Link(Link* nextval =NULL)
   { next = nextval; }
};
```

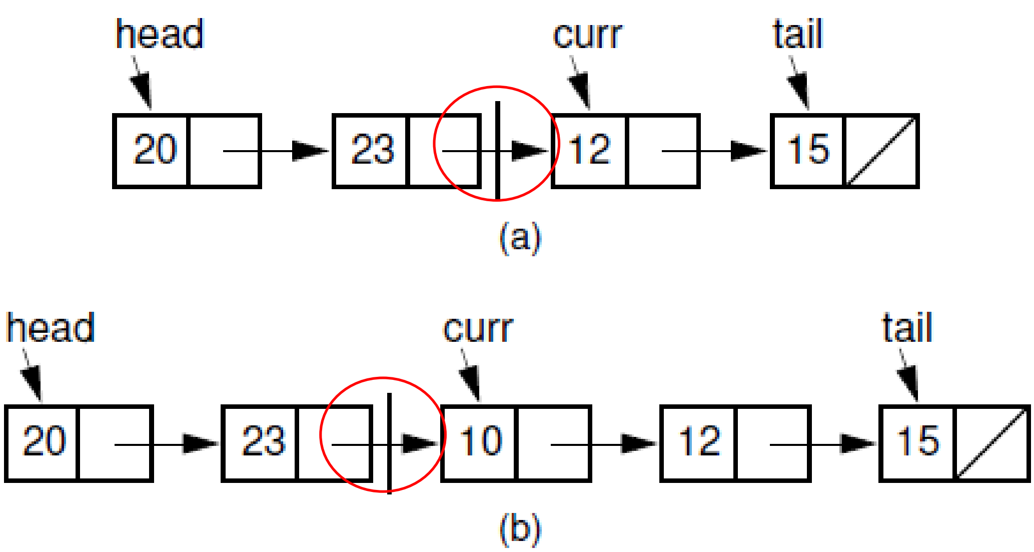
List: Pointer Implementation

- A linked list with 4 elements
  - Head pointer – for scanning the whole list
  - Tail pointer – to speed up “append” operation
  - Curr pointer – pointing to the current element
  - Value cnt – store the length of the list



List: Pointer Implementation

- Insertion
  - With *curr* points to the node after the current position

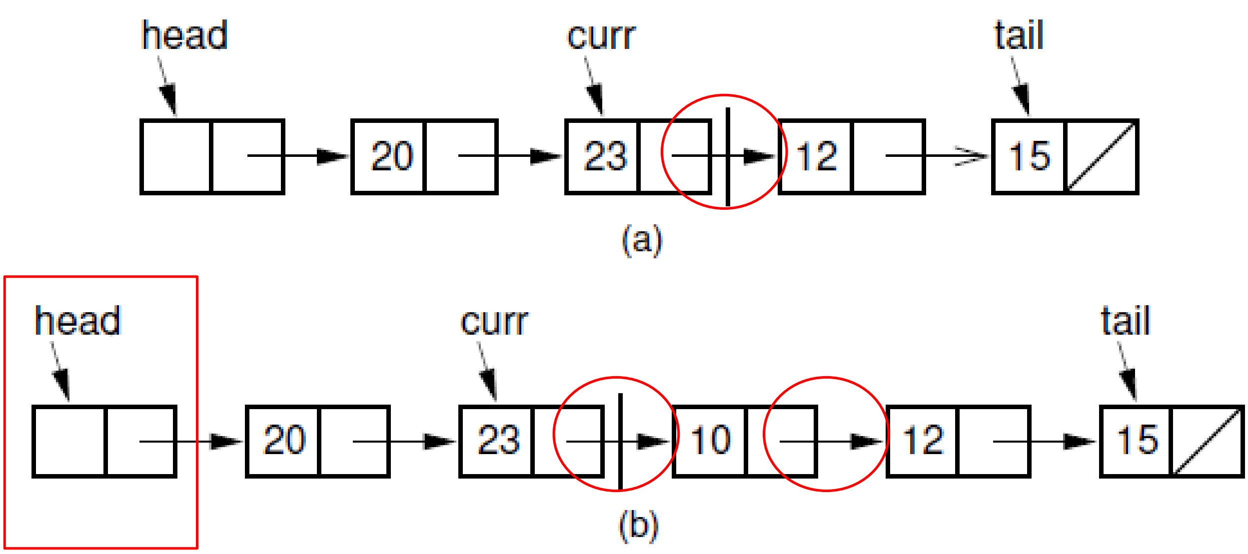


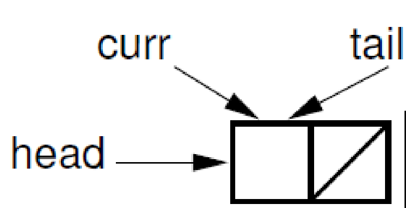
(a)

(b)

List: Pointer Implementation

- Insertion
  - With *curr* points to the node preceding the current position



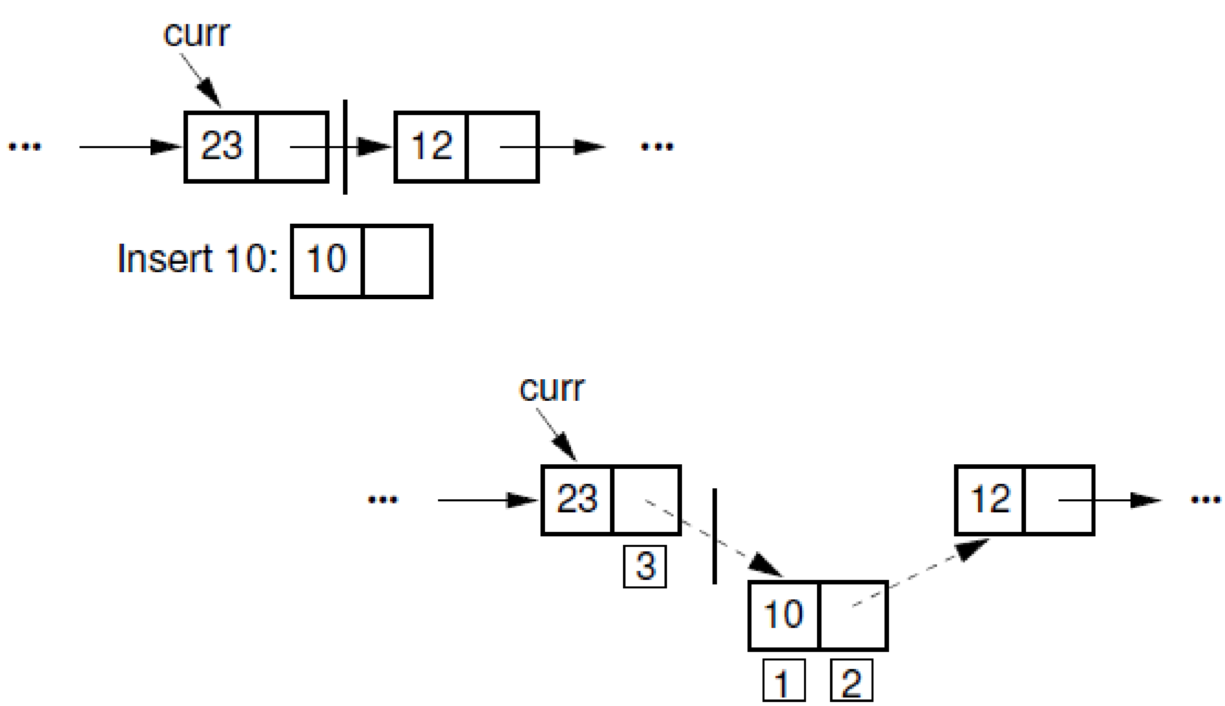


List: Pointer Implementation

```
//Class LList
//inherit the abstract class List
template <typename E> class LList: public List<E>{
private:
    Link<E>* head; //Pointer to list header
    Link<E>* tail; //Pointer to last element
    Link<E>* curr; //access to current element
    int cnt;       //Size of list
    void init() //used by constructor
    {  curr = tail = head = new Link<E>; cnt =0;}
    void removeall() //used by deconstructor
    {   while(head != NULL) {
            curr=head; head=head->next; delete curr;
        }
    }
}
```

List: Pointer Implementation

- Linked List – Insertion
  - Three-step insertion process
    - Create a new list node, store the new element
    - Set the next field of the new node
    - set the next field of the node pointed by <u>*curr*</u>



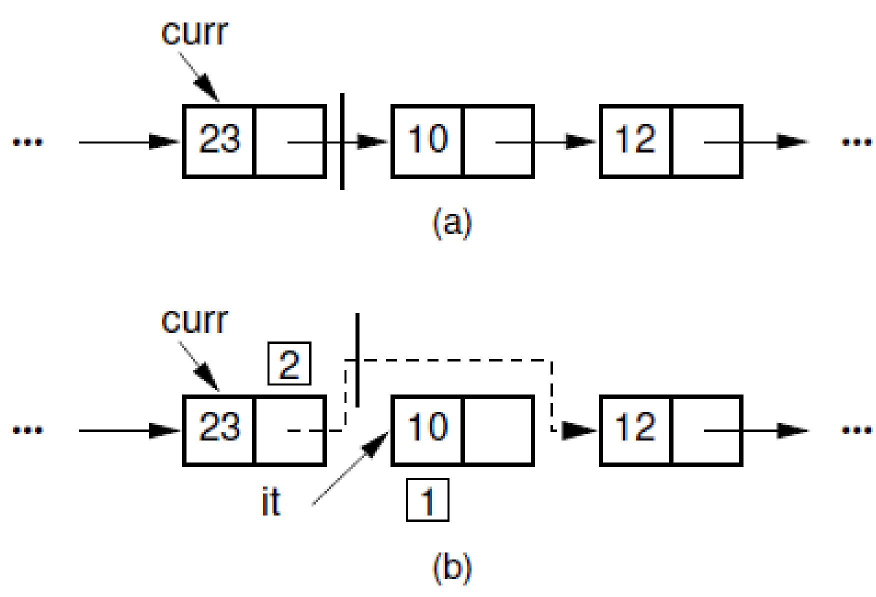
List: Pointer Implementation

```
//Insert a node to current position
  public:
    void insert(const E& it) {
        curr->next = new Link<E>(it, curr->next);
        if (tail == curr) tail = curr->next; //new tail
        cnt++;
    }
    //Append a node at the tail of the list
    void append(const E& it) {
        tail = tail->next = new Link<E>(it, NULL);
        cnt++;
    }
```

Time cost – Θ(1)

List: Pointer Implementation

- Linked List – Removal
  - Removing a node only requires to redirect some pointers around the node to be deleted.
  - Remember to reclaim the space occupied by the deleted node



List: Pointer Implementation

```
// Remove and return current element
 E remove() {
      E it = curr->next->element;
      Link<E> *ltemp = curr->next;
      if (tail == curr->next)
              tail =curr; // Reset tail
      curr->next = curr->next->next; //remove element
      delete ltemp;  //reclaim space
      cnt--;          //decrement the list size
      return it;
 }
```

Time cost – $\Theta(1)$

List: Pointer Implementation

- Linked List – Position Ops

```
//Next – move curr one pos toward the tail
 void next() { // no change if already at end
      if (curr != tail) { curr = curr->next;}
 }
//Prev – move curr one pos toward the head
 void prev() {
    if (curr == head) return; // No previous element
    Link<E>* temp = head;
   //march down list until the previous element
   while (temp->next!=curr) temp=temp->next;
   curr = temp;
}
```

Time cost: $\Theta(1)$ for next;

$\Theta(n)$ for prev in the average and worst cases.

List: Pointer Implementation

- C++ STL – List
  - a doubly linked list implementation of the List ADT
    - How to implement the list ADT with pointers where Pre method takes $\Theta(1)$ time?
  - Textbook section 3.3 list & Code list.h and TestList.cpp

Comparison of List Implementations

Array-Based List		Linked List	
Predetermine the size before allocation.		Space is allocated on demand; No limit to the element number.	√
No waste space for an individual element.	√	Require to add an extra pointer to every list node.	
Random access and Prev takes $\Theta(1)$ time	√	Random access and Prev takes $\Theta(n)$ time	
Insertion and deletion takes $\Theta(n)$ time.		Insertion and deletion takes $\Theta(1)$ time.	√

Comparison of List Implementations

- linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown.

- Array-based lists are generally more space efficient when the user knows in advance approximately how large the list will become.

Comparison of List Implementations

- Comparison formula
  - The number of element currently in the list – n;
  - The size of a pointer – P
  - The size of a data element – E
  - The maximum number of elements in the array – D

- The array-based list requires space DE

- The linked list requires space n(P+E)

When n> DE/(P+E), the array-based list is more space efficient!

Exercise

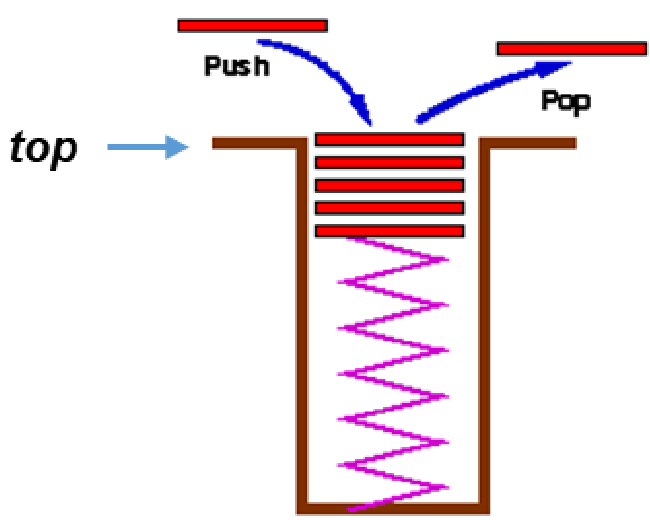
- Determine the break-even point for a linked list being more efficient than an array-based list
  - The data field is 2 bytes, a pointer is 4 bytes, the array has 30 elements
    - $n < DE/(P+E) = 2*30/(2+4) = 10$
  - The data field is 8 bytes, a pointer is 4 bytes, the array has 30 elements
    - $n < DE/(P+E) = 8*30/(8+4) = 20$
  - The data field is 32 bytes, a pointer is 4 bytes, the array has 40 elements
    - $n < DE/(P+E) = 32*40/(32+4) = 35.555$

Stack

What is Stack

- A list for which Insert and Delete are allowed only at one end of the list (the top)
  - the implementation defines which end is the "top"
  - LIFO – Last in, First out

- Push: Insert element at top
- Pop: Remove and return top element (aka TopAndPop)
- IsEmpty: test for emptyness



a tray stack

Two Basic Implementations of Stacks

- Linked List

- Array-based
  - The k items in the stack are the first k items in the array.

Stack ADT

```
// The stack ADT
template <typename E> class Stack {
private:
   void operator=(const Stack&) {} //Protect assignment
   Stack(const Stack&) {} //Protect copy assignment

public:
     // Push an element onto the top of the stack.
     virtual bool push(const E& it) = 0;

    // Remove the element at the top of the stack.
    virtual E pop() = 0;

   // Return a copy of the top element
   virtual const E& topValue() const = 0;
};
```

Array-Based Stacks

- Array-based stack implementation
  - The stack must be declared of fixed size
  - A simplified version of the array-based list

```
Template <typename E> class Astack: public Stack<E> {
private:
  int maxSize;     // Maximum size of stack
  int top;      // Index for top element (free position)
  E *listArray; // Array holding stack elements

public:
      AStack(int size=defaultSize) //Constructor
      { maxSize=size; top=0; listArray=new E[size]; }
      ~Astack() { delete [] listArray; } //Destructor
```

Which end of the array shall be the top of the stack?

Array-Based Stacks (III)

- Make the tail of the array be the top of the stack
  - Pushing an element onto the stack by appending it to the tail of the list
  - The cost for each **push** and **pop** operation is simply $\Theta(1)$.

- Setting of **top**
  - The array index of the first free position in the stack
    - An empty stack has top set to 0.
  - Push: first insert the element, then increment top
  - Pop: first decrement top, then removes the top element;
    - Pay attention to the order of two operations

Array-Based Stacks (II)

```
void clear() { top=0; }     //Reinitialize

void push(const E& it) { // put "it" on stack
  Assert(top != maxSize, "Stack is full");
  listArray[top++] = it; }

E pop() {//pop top element
  Assert(top != 0, "Stack is empty");
  return listArray[--top]; }

const E& topValue() const {    //return top element
  Assert(top != 0, "Stack is empty");
  return listArray[top-1]; }
```

Linked Stacks

- Elements are inserted and removed only from the head of the list

```
//A linked stack
template <typename E> class LStack: public
Stack<E> {
private:
          Link<E>* top;     //pointer to first element
          int size;         //number of elements
public:
    LStack(int sz = defaultSize){     //Constructor
        top = NULL; size = 0;
    }
    ~LStack()  { clear(); }           //Destructor
```

Linked Stacks (II)

```
void clear() {                     //reinitialize
   while (top != NULL) {           //delete link nodes
      Link<E>* temp = top; top = top->next; delete temp;
   }
   size = 0;
}

void push(const E& it) {           //put “it” on the stack
    top = new Link<E>(it, top); size++;
}

E pop() {                          //remove “it” from stack
   Assert(top != NULL, “Stack is empty”);
   E it = top->element;
   Link<E>* ltemp = top->next;
   delete top;  top = ltemp;  size--; return it;
}

const E& topValue() const {   //return top value
   Assert(top != 0, “Stack is empty”);
   return top->element;
}
```

Linked Stacks (III)

- No need to have a head node
  - No special code is required for lists of zero or one elements.
- A pointer **top** points to the first link node
  - Push: first modifies the **next** field of the newly created node to point to the top of the stack, then sets **top** to point to the new node
  - Pop: set **top** to point to the **next** link of the old top node; the old top node is freed and its element value is returned.

Comparison of Array-Based and Linked Stacks

	Array-Based Stack	**Linked Stack**
Implementation	Take the end of array as the top of stack	Take the head of linked list as the top of stack
Time cost	Constant time for push, pop, topValue; Constant time for clear	Constant time for push, pop and topValue; Linear time for clear
Space cost	Waste some space when the stack is not full - Overflow possible	Require the overhead of a link field for every element

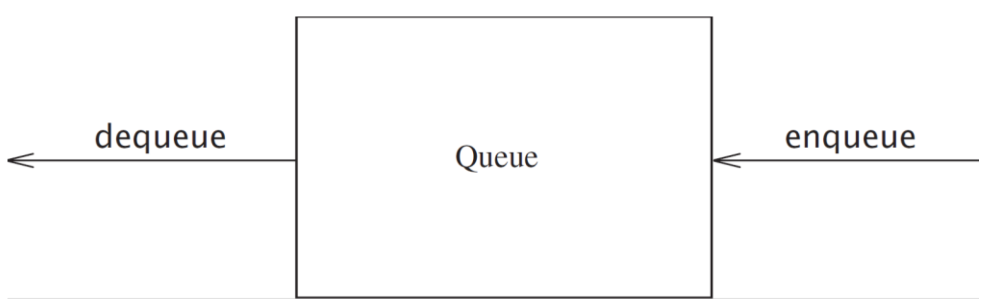
Q: How to implement two stacks using a single array?

Queue

图片来自互联网

What is Queue

- In a queue, elements may only be inserted from one end (back) of the list and removed from the other end (front) of the list
  - First-In, First-Out
  - Enqueue: insert an element at the back
  - Dequeue: remove an element from the front



Queue ADT

```
template <typename E> class Queue {
private:
   void operator =(const Queue&) {}
    Queue(const Queue&) {};
public:
   Queue() {}
   virtual ~Queue() {}

   virtual void clear() = 0;
    virtual void enqueue(const E&) = 0;
    virtual E dequeue()  = 0;
    virtual const E& frontValue() const = 0;
    virtual int length() const = 0;
}
```

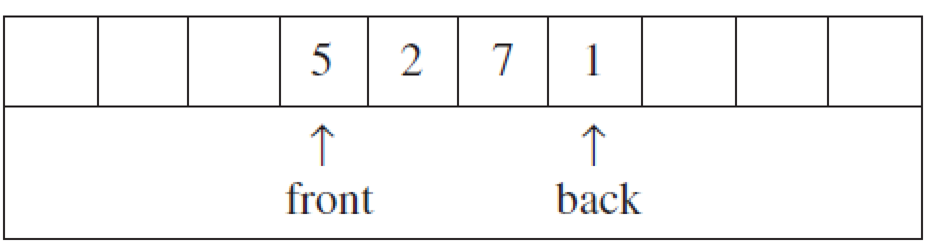
Two Implementations of Queue

- The array-based queue
- The linked queue

Queue: Array Implementation

```
//Array Implementation for Queue

template <typename E> class AQueue: public Queue<E> {
  private:
    int maxSize; // Maximum size of queue
    int front; // Index of front element
    int rear; // Index of back element
    E *listArray; // Array holding queue elements
  int currSize; //length of queue
  …
};
```



Array-Based Queues

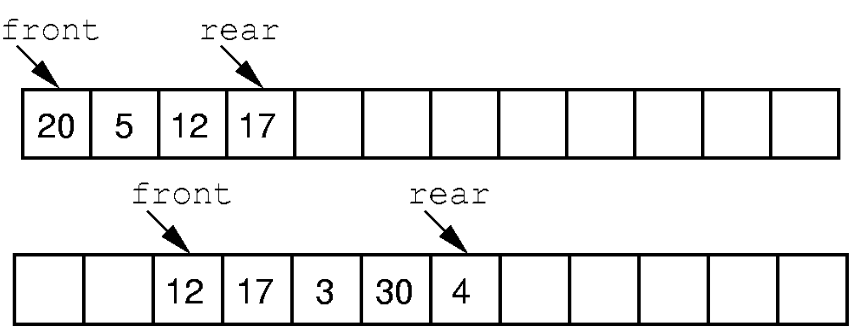
- Suppose we store n elements in the first n positions of the array
  - If we choose position n-1 in the array as the front of the queue
    - **Dequeue** requires only $\Theta(1)$ time, but **enqueue** costs $\Theta(n)$ time
  - If we choose position 0 in the array as the front of the queue
    - **Dequeue** costs $\Theta(n)$ time, while **enqueue** costs $\Theta(1)$ time

Array-Based Queues

- An efficient and tricky implementation
  - The queue is still required to be stored in contiguous array positions
  - The queue position can **drift** within the array

Array-Based Queues

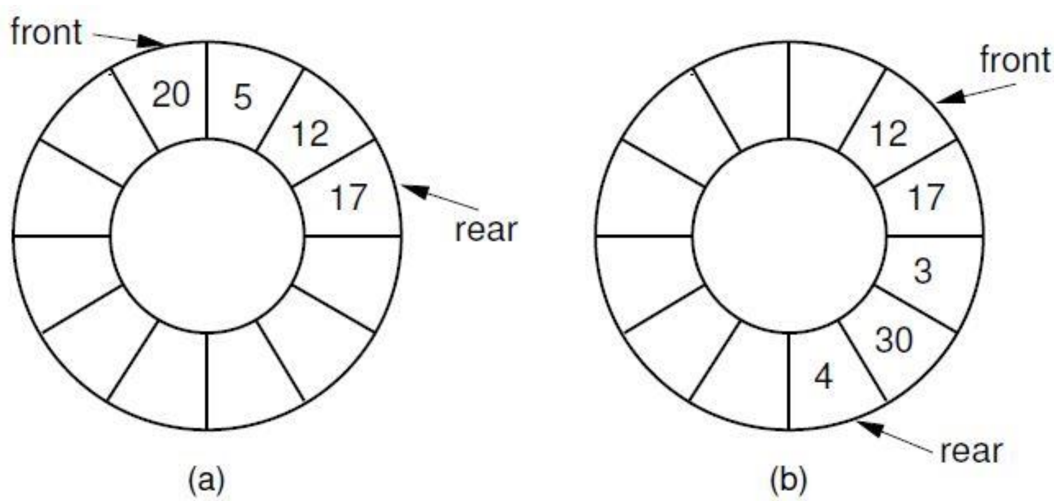
- Drifting queue
  - The **front** of the queue is initially at **position 0** of the array
  - The elements are added to successively higher-numbered positions
  - When elements are removed, the front index increases
  - **Both** enqueuer and dequeuer cost $\Theta(1)$ time



Problem?

Array-Based Queues

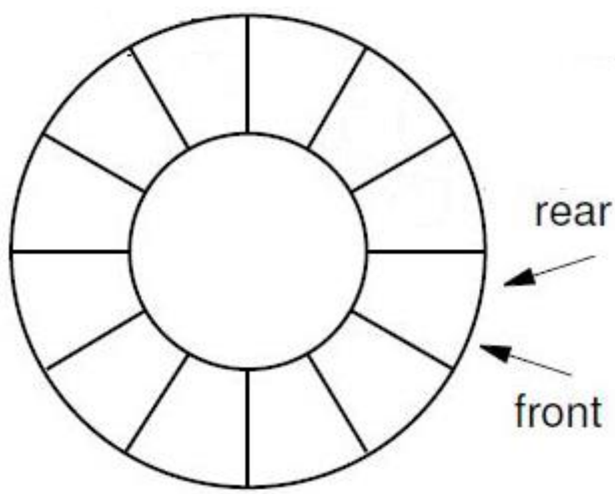
- Circular queue



- Easily implemented using the modulus operator
  - Position maxSize-1 immediately precede position 0

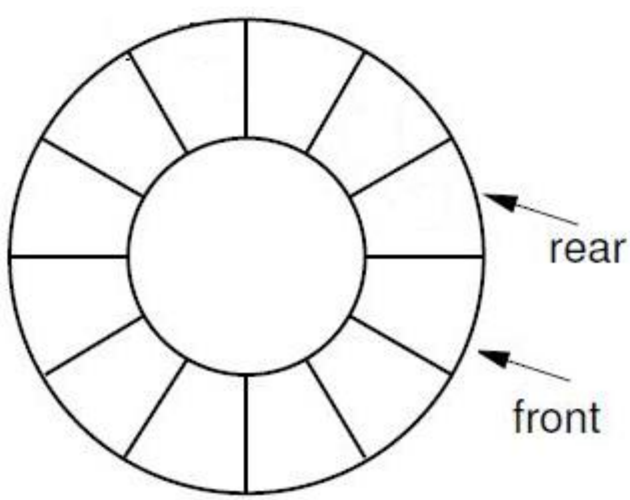
Array-Based Queues

- Circular queue
  - How to recognize whether the queue is empty or full?



Array-Based Queues

- Circular queue
  - How to recognize whether the queue is empty or full?



Array-Based Queues

- When **front = rear**, there has one element in the queue

- When **front** is one larger than **rear**, the queue is empty or full?
  - Solution 1: explicitly keep a count of the number of elements in the queue

  - Solution 2: make the array be of size n+1 and only allow n elements to be stored.
    - **front** = **rear**+1, the queue is empty
    - **front = rear**+2, the queue is full.

Array-Based Queues

```
//Array-based Implementation (solution 2)
template <typename E> class AQueue: public Queue<E> {
  private:
  int maxSize; // Maximum size of queue
  int front; // Index of front element
  int rear; // Index of rear element
  E *listArray; //Array holding queue elements
  public:
  AQueue(int size =defaultSize) {
  // Constructor- Make list array one position
  // larger for empty slot
    maxSize = size+1;
    rear = 0; front = 1;
    listArray = new E[maxSize];
  }
  ˜AQueue() { delete [] listArray; }
```

Array-Based Queues

```
//reinitialize
void clear() {rear = 0;  front = 1;}
void enqueue(const E& it) {//put “it” in queue
  Assert(((rear+2) % maxSize) != front,
          “Queue is full”);
  rear = (rear+1)%maxSize;
  listArray[rear] = it;
}
E dequeue() {//take element out
  Assert(length() != 0, “Queue is empty”);
  E it = listArray[front];
  front =(front+1)%maxSize;
  return it;
}
```

Array-Based Queues

```
//get front value
const E& frontValue() const {
  Assert(length()!=0, "Queue is empty");
  return listArray[front];
}

//return length
virtual int length() const  {
  return ((rear+maxSize)-front+1)%maxSize;
}
```

Linked Queues

- A straightforward adaptation of the linked list

- Structures
  - Use a header node
  - The **front** pointer points always points to the header node
  - The **rear** pointer points to the last link node in the queue

- Operations
  - **Enqueue**: places the new element in a link node at the end of the linked list, advances **rear** to point to the newly-inserted node
  - **Dequeue**: removes and returns the first element of the list

Comparison of Array-Based and Linked Queues

- Time cost
  - All member functions for both implementations require constant time **$\Theta(1)$**
- Space cost
  - For array-based queues, there are some space waste if the queue is not full.
  - For linked queues, there are overhead of link field in each element.

Homework

- Exercise 3.28, 3.30, and
- Show how to implement **two** stacks in one array
- Deadline: to be confirmed.