Data Structure and Algorithms

Chapter 4 Stack and Queue

Dr. Zhiqiang Liu

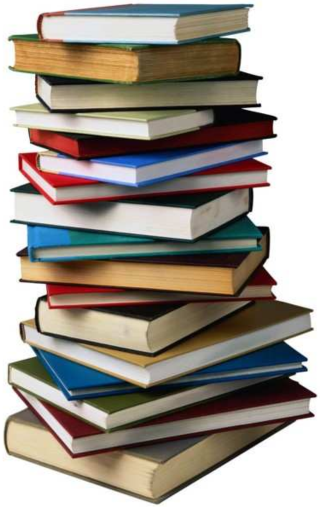
School of Software and Microelectronics, Northwest Polytechnical University

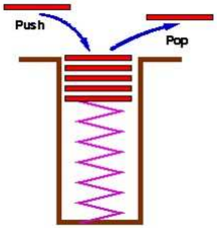
Outline

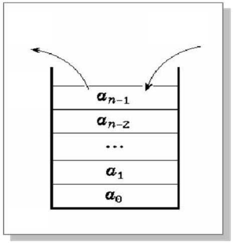
Next Section

Next Subsection

Example of Stack, what else?





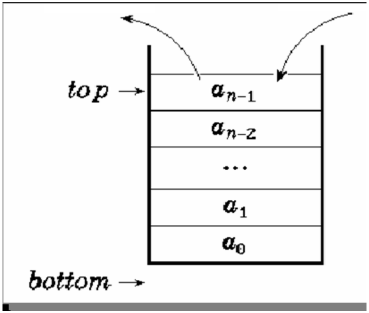


Example of Stack

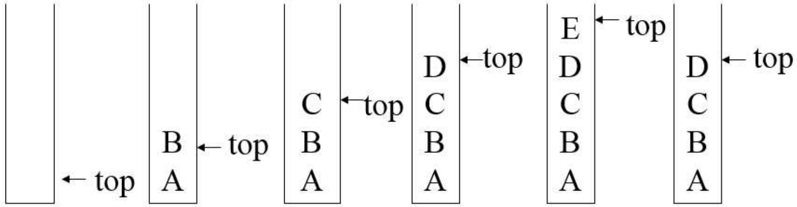
Concepts

- **Top**
  - The specific end of linear list at which elements can inserted and removed.
- **Bottom**
  - Another end of the stack.

Removal (Pop)   Insertion (Push)



Examples of Push and Pop



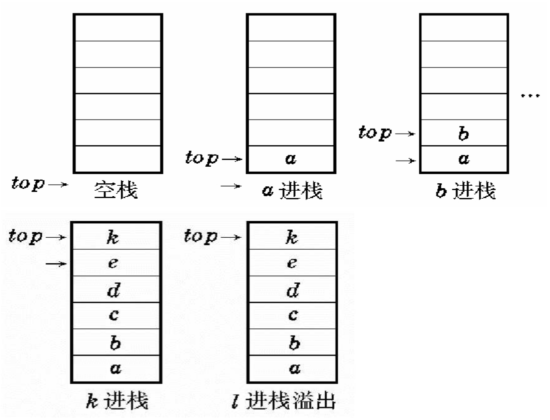
Inserting and deleting elements in a stack

Next Subsection

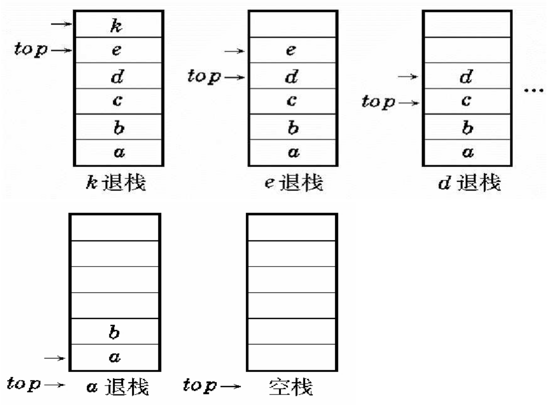
4.1.2 ADT of Stacks

```
template <class Type> class Stack {
public:
    Stack ( int=10 );
    void Push ( const Type & item);
    Type Pop ( );
    Type GetTop ( );
    void MakeEmpty ( );
    int IsEmpty ( ) const;
    int IsFull ( ) const;
}
```

Sequential Stack Implementation (Using Array): Push Demo



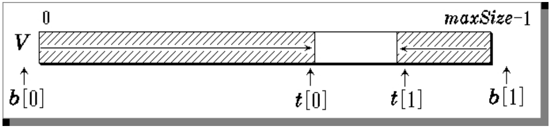
Sequential Stack Implementation (Using Array): Pop Demo



```
#include <assert.h>
template <class Type> class Stack {
public:
    Stack ( int=10 );
    ~Stack ( ) { delete [ ] elements; }
    void Push ( const Type & item );
    Type Pop ( );
    Type GetTop ( );
    void MakeEmpty ( ) { top = -1; }
    int IsEmpty ( ) const { return top == -1; }
    int IsFull ( ) const { return top == maxSize-1; }
private:
    int top;
    Type *elements;
    int maxSize;
}

template <class Type> Stack<Type>::
Stack ( int s ) : top (-1), maxSize (s) {
    elements = new Type[maxSize];
    assert ( elements != 0 );
}

template <class Type> void Stack<Type>::
Push ( const Type & item ) {
    assert ( !IsFull ( ) );
    elements[++top] = item;
}

template <class Type> Type Stack<Type>::
Pop ( ) {
    assert ( !IsEmpty ( ) );
    return elements[top--];
}

template <class Type> Type stack<Type>::
GetTop ( ) {
    assert ( !IsEmpty ( ) );
    return elements[top];
}
```

Two stacks use the same memory block

Stack 1

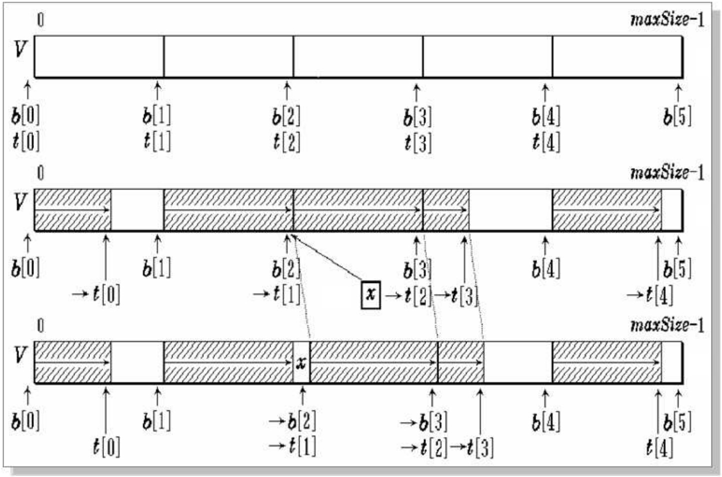


Stack 2

- Initialization.
- Push entry into stack 1 or stack 2.
- Pop entry from stacks.
- Critical conditions
  - Empty
  - Full
  - Overflow

Question?

- n stacks in the same array

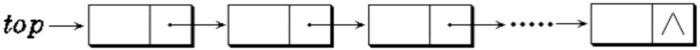


Initialization

Push x onto stack 2

After pushing x onto stack 2

Stack Implementation Use Linked list



```
template <class Type> class Stack;

template <class Type> class StackNode {
friend class Stack<Type>;
private:
    Type data;
    StackNode<Type> *link;
    StackNode ( Type d = 0, StackNode<Type> *l = NULL )
        : data ( d ), link ( l ) { }
};
template <class Type> class Stack {
public:
    Stack ( ) : top ( NULL ) { }
    ~Stack ( );
    void Push ( const Type & item);
    Type Pop ( );
    Type GetTop ( );
    void MakeEmpty ( );
    int IsEmpty ( ) const
         { return top == NULL; }
private:
    StackNode<Type> *top;
}
template <class Type> Stack<Type>::
~Stack ( ) {
    StackNode<Type> *p;
    while ( top != NULL ) {
        p = top;  top = top->link;  delete p;
    }
}
template <class Type> void Stack<Type>::
Push ( const Type &item ) {
    top = new StackNode<Type> ( item, top );
}
template <class Type> Type Stack<Type>::
Pop ( ) {
    assert ( !IsEmpty ( ) );
    StackNode<Type> *p = top;
    Type retvalue = p->data;
    top = top->link;
    delete p;   return retvalue;
}
template <class Type> Type Stack<Type>::
GetTop ( ) {
    assert ( !IsEmpty ( ) );
    return top->data;
}
```

Next Subsection

4.1.3 Application 1: Bracket Matching

- Requirement
  - Develop a program to check that brackets are correctly matched in an input text file.
  - The brackets are limited in {, }, (,), [, and].
- Methods
  - Read a single line of characters and ignore all input other than bracket characters.
  - Our program need only loop over the input characters, until either a bracketing error is detected or the input file ends.

4.1.3 Application 1: Bracket Matching

- Algorithm:
  - Read the file character by character. Each opening bracket (, [, or { that is encountered is considered as unmatched and is stored until a matching bracket can be found. Any closing bracket),], or } must correspond, in bracket style, to the last unmatched opening bracket, which should now be retrieved and removed from storage. Finally, at the end of the program, we must check that no unmatched opening brackets are left over.

$$a * \{[b + c/(e - f)] - x\} - 1$$

$$a * \{[b + c/(e - f\} + 2)] - x\} - 1$$

Bracket Matching Program

```
int main() {
   Stack openings;
   char symbol;
   bool is_matched = true;
  while (is_matched && (symbol = cin.get()) != '\n') {
      if (symbol == '{' || symbol == '(' || symbol == '[
         ')
         openings.push(symbol);
      if (symbol == '}' || symbol == ')' || symbol == ']
         ') {
         if (openings.empty()) {
            cout << “Unmatched closing bracket ” <<
                symbol << “ detected”. << endl;
            is_matched = false;
         } else {
            char match;
            match = openings.pop();
            is_matched=(symbol == '}' && match == '{')
                || (symbol == ')' && match == '(')
                || (symbol == ']' && match =='[');
            if (!is_matched)
                cout << "Bad␣match␣" << match << symbol
                    << endl;
        }
      }
   }
   if (!openings.empty())
      cout << “Unmatched opening bracket(s) detected”.
          << endl;
}
```

Next Subsection

Application 2: Reverse Polish calculator

- Expression
  - Infix: a+b*(c-d)-e/f
  - Postfix (Reverse Polish Notation): abcd-*+ef/-
  - Prefix (Polish Notation): -+a*b-cd/ef
- Operands: a, b, c, d, e, f
- Operators: +, -, *, /,
- Delimiter: (,)

Expression Evaluation

How to calculate the value of the expression?

Infix expression

a+b*(c-d)-e/f

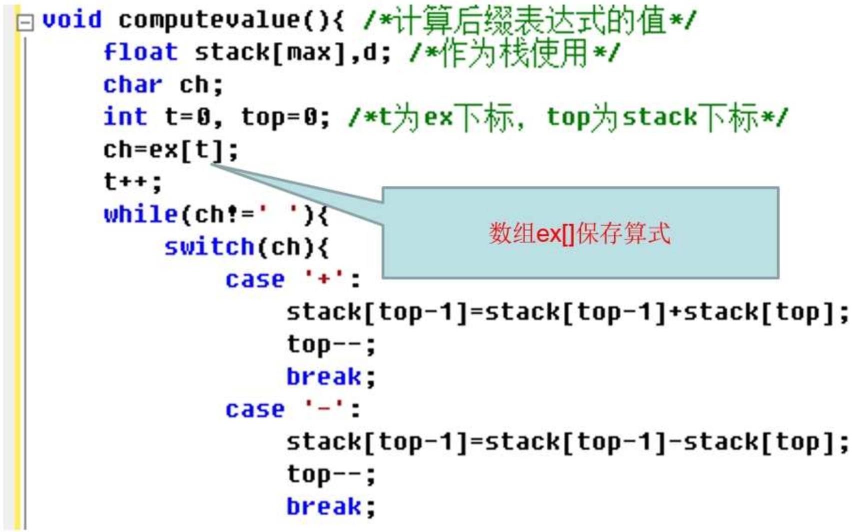
Prefix expression

abcd-*+ef/-

-+a*b-cd/ef

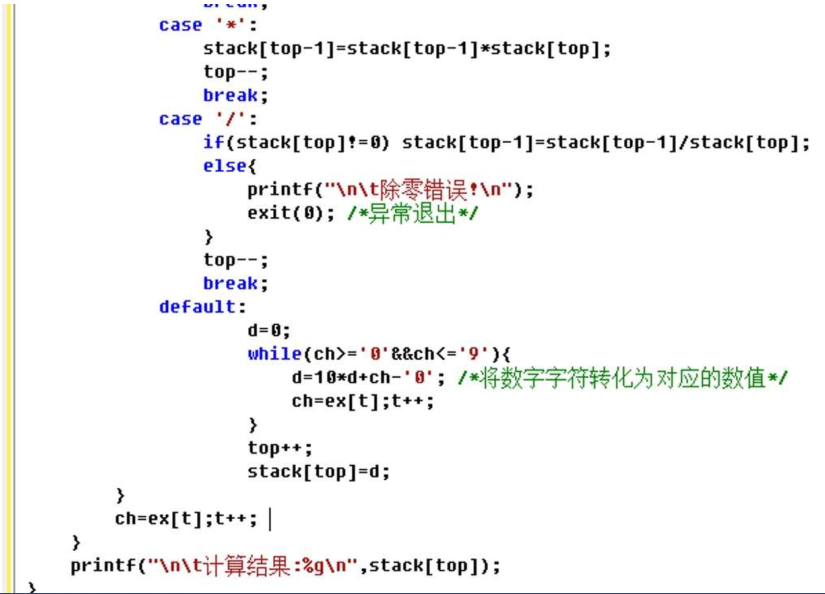
Postfix expression

Reverse Polish calculator



```
void computevalue(){ /*计算后缀表达式的值*/
    float stack[max],d; /*作为栈使用*/
    char ch;
    int t=0, top=0; /*t为ex下标, top为stack下标*/
    ch=ex[t];
    t++;
    while(ch!=' '){
        switch(ch){
            case '+':
                stack[top-1]=stack[top-1]+stack[top];
                top--;
                break;
            case '-':
                stack[top-1]=stack[top-1]-stack[top];
                top--;
                break;
```

4.1.4 Application 2: Reverse Polish calculator



```
                break;
            case '*':
                stack[top-1]=stack[top-1]*stack[top];
                top--;
                break;
            case '/':
                if(stack[top]!=0) stack[top-1]=stack[top-1]/stack[top];
                else{
                    printf("\n\t除零错误!\n");
                    exit(0); /*异常退出*/
                }
                top--;
                break;
            default:
                    d=0;
                    while(ch>='0'&&ch<='9'){
                        d=10*d+ch-'0'; /*将数字字符转化为对应的数值*/
                        ch=ex[t];t++;
                    }
                    top++;
                    stack[top]=d;
        }
        ch=ex[t];t++;
    }
    printf("\n\t计算结果:%g\n",stack[top]);
}
```

Next Subsection

4.1.5 Application 3: Infix Expression to Postfix

- Expression Conversion
  - Infix: a+b*(c-d)-e/f
  - Postfix (Reverse Polish Notation): abcd-*+ef/-
- Components
  - Operands: a, b, c, d, e, f
  - POperators: +, -, *, /,
  - Delimiter: (,), #

4.1.5 Application 3: Infix Expression to Postfix

#a+b*(c-d)-e/f# $\Rightarrow$ abcd-*+ef/-

A specific stack OPTR is used to store temporary operators, such as +, *, and (in above expression. isp (**in stack priority**): the priority of operator at the top of OPTR stack. icp (**incoming priority**): the new incoming operator priority.

If isp < icp, push the incoming operator into OPTR
If isp > icp, output and pop the top of OPTR
If isp = icp, scan next item and pop the top of OPTR

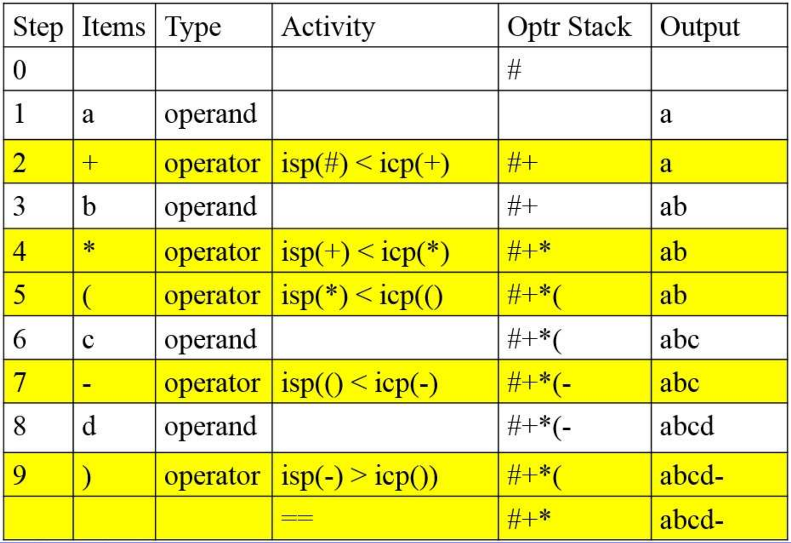
Priority of operators

θ_1 -- isp θ_2 -- icp

$\theta_1 \backslash \theta_2$	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

Example

Infix expression: a+b*(c-d)-e/f



Step	Items	Type	Activity	Optr Stack	Output
0				#	
1	a	operand			a
2	+	operator	isp(#) < icp(+)	#+	a
3	b	operand		#+	ab
4	*	operator	isp(+) < icp(*)	#+*	ab
5	(	operator	isp(*) < icp(()	#+*(	ab
6	c	operand		#+*(	abc
7	-	operator	isp(() < icp(-)	#+*(-	abc
8	d	operand		#+*(-	abcd
9	)	operator	isp(-) > icp())	#+*(	abcd-
			==	#+*	abcd-

Example

10	-	operator	isp(*) > icp(-)	#+	abcd-*
			isp(+) > icp(-)	#	abcd-*+
			isp(#) < icp(-)	#-	abcd-*+
11	e	operand		#-	abcd-*+e
12	/	Operator	isp(-) < icp(/)	#-/	abcd-*+e
13	f	Operand		#-/	abcd-*+ef
14	#	Operator	isp(/) > icp(#)	#-	abcd-*+ef/
			isp(-) > icp(#)	#	abcd-*+ef/-
			==, end		

#a+b*(c-d)-e/f# $\Rightarrow$ abcd-*+ef/-

Next Subsection

4.1.6 Summary and QUIZ

4.1.6 Summary and QUIZ

1、设顺序栈S，元素1-6依次进栈，出栈顺序为2、3、4、6、5、1，则栈的容量至少为多少?

2、设有编号为1至6的6个元素，将这些元素按照编号由小到大的顺序压栈，且每个元素只能压栈一次。
1）能否得到435612，325641，154623和135426的出栈序列？如果能，请说明如何得到（即写出元素进栈和出栈的顺序）。如果不能，请说明原因。
2）如果元素按照123的顺序进栈，那么这三个元素可能的出栈序列是哪几种?

Next Section

Next Subsection

4.2.1 Definition of Queues

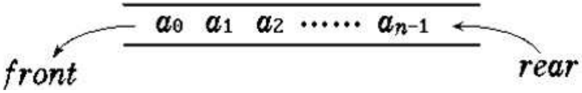
- A queue is a special linear list.
- A queue is a list in which all addition to the list are made at one end, and all deletions from the list are made at the other end.
- FIFO structure: The first entry which is added is the first one that will be removed.
  (First In First Out)

Example

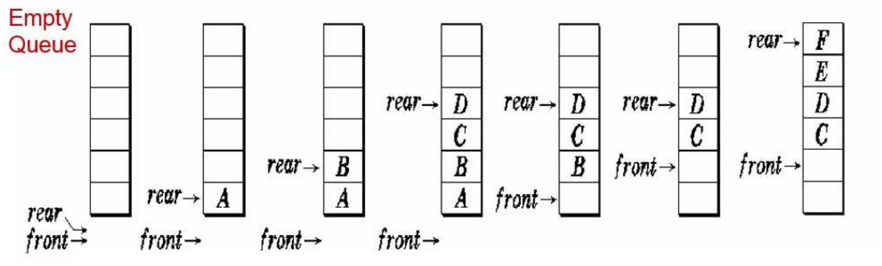
A queue is a waiting line, like a people waiting to get on the train, where the first person in line is the first person boarded.

Concepts

- **front**
  - The specific end of linear list at which elements can removed.
  - The entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front (head) entry of the queue.
- **rear**
  - The specific end of linear list at which elements can added or inserted.
  - The one most recently added, is called the rear (tail) entry of the queue.



Examples of Addition and Deletion



- **Alternative methods**
  - insert, append, enqueue
  - delete, serve, dequeue

Next Subsection

ADT of Queues

```
template <class Type> class Queue {
public:
   Queue ( int=10 );
   void EnQueue ( const Type & item);       Key functions
   Type DeQueue ( );
   Type GetFront ( );
   void MakeEmpty ( );
   int IsEmpty ( ) const ;
   int IsFull ( ) const;
}
```

Problem of linear implementation in array

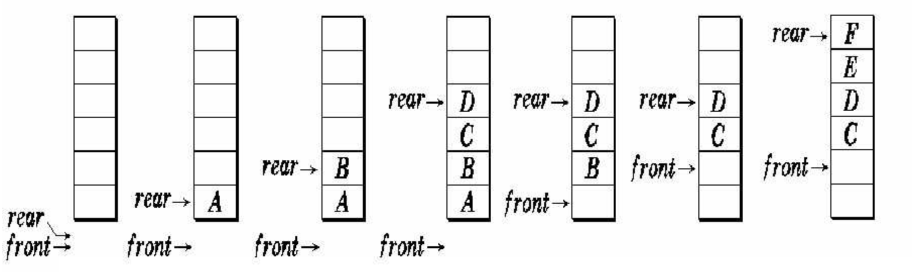
For example

rear=5, front=1

the actual length of queue is 4

however, no new entry could be appended into queue.

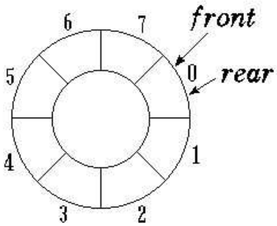
Why?



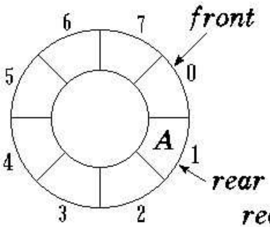
Problem of linear implementation in array

- **Problem**
  - front and rear indices will move to the high end of the array after several Enqueue and Dequeue operations
- **Solution**
  - keep the front index always in the first location of the array — long time taken
  - thinking of the array as a circle rather than a straight line

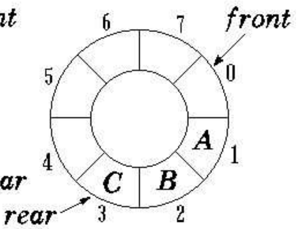
Solution



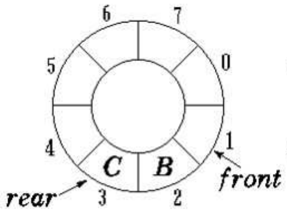
Empty Queue



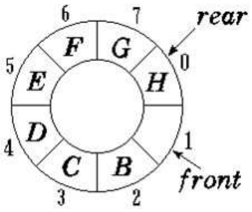
Enqueue A



Enqueue BC



Dequeue A

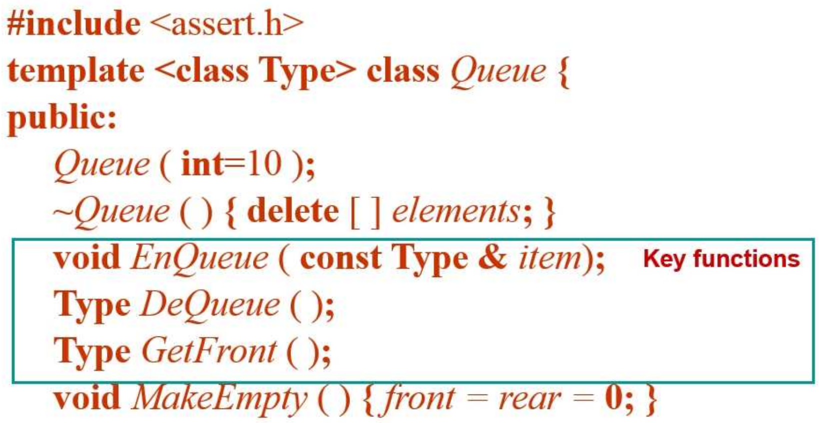


Enqueue DEFGH

Next Subsection

Circular Implementations of Queues

Class declaration

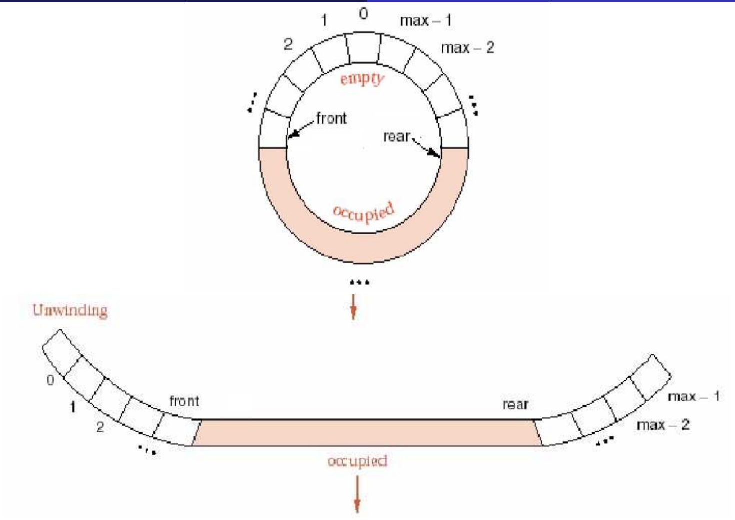


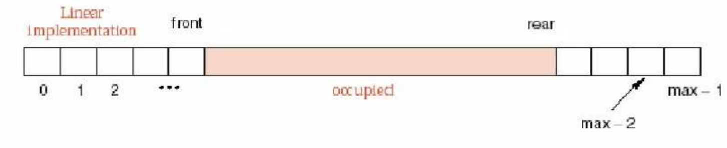
```
#include <assert.h>
template <class Type> class Queue {
public:
    Queue ( int=10 );
    ~Queue ( ) { delete [ ] elements; }
    void EnQueue ( const Type & item);
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( ) { front = rear = 0; }
```

Key functions

Circular Implementations of Queues

```
    int IsEmpty ( ) const
       { return front == rear; }
    int IsFull ( ) const
       { return (rear+1) % maxSize == front; }
    int Length ( ) const
       { return (rear-front+maxSize) % maxSize;}
private:
    int rear, front;
    Type *elements;
    int maxSize;
}
```


1
0
max – 1
2
max – 2
empty
front
rear
occupied
Unwinding
0
1
2
front
rear
max – 1
max – 2
occupied


Linear implementation
front
rear
0
1
2
...
occupied
max – 1
max – 2

Circular array in C++

Equivalent methods to increment an index **i** in a circular array:

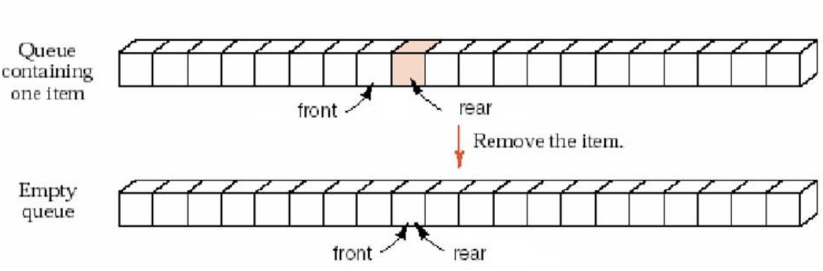
```cpp
i = ((i+1) == max) ? 0 : (i+1);
```

```cpp
if ((i+1) == max) i=0 ; else i=i+1;
```

```cpp
i = (i+1) % max;
```

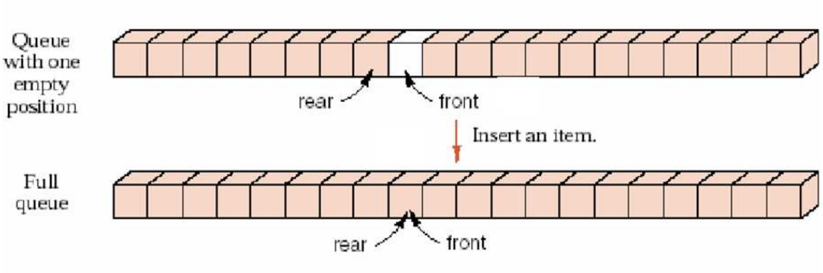
Boundary conditions

- **Empty**
  - front == rear;
  - due to front++

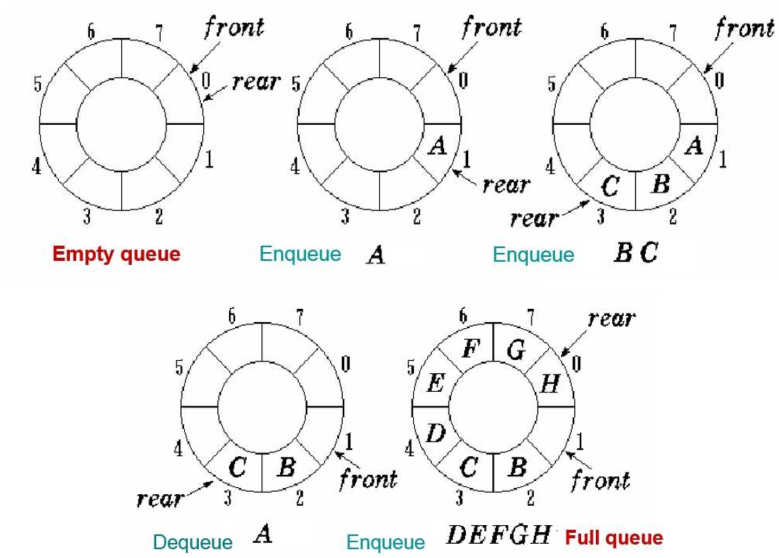


Boundary conditions

- **Full**
  - rear == front
  - due to rear++



Boundary conditions



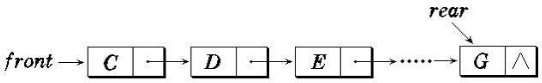
Circular array in C++

```
template <class Type> Queue<Type>::
Queue ( int sz ) : front (0), rear (0), maxSize (sz) {
    elements = new Type[maxSize];
    assert ( elements != 0 );
}

template <class Type> void Queue<Type>::
EnQueue ( const Type & item ) {
     assert ( !IsFull ( ) );
     rear = (rear+1) % MaxSize;
     elements[rear] = item;
}




template <class Type> Type Queue<Type>::
DeQueue ( ) {
     assert ( !IsEmpty ( ) );
```

```
        front = ( front+1) % MaxSize;
        return elements[front];
}

template <class Type>  Type Queue<Type>::
GetFront ( ) {
        assert ( !IsEmpty ( ) );
        return elements[front];
}
```

Next Subsection

Linked Queue



```
template <class Type> class Queue;
template <class Type> class QueueNode {
friend class Queue<Type>;
private:
    Type data; //Data element
    QueueNode<Type> *link; //pointer to next item
    QueueNode ( Type d=0, QueueNode *l=NULL ) : data (d)
        , link (l) { }
};



template <class Type> class Queue {
public:
    Queue ( ) : rear ( NULL ), front ( NULL ) { }
    ~Queue ( );
    void EnQueue ( const Type & item );
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( ); //same as function ~Queue( )
    int IsEmpty ( ) const { return front == NULL; }
private:
    QueueNode<Type> *front, *rear; //Pointer indicators
};

template <class Type> Queue<Type>::
~Queue ( ) {
    //Destructor
    QueueNode<Type> *p;
    while ( front != NULL ) {
        //Release every node step by step
        p = front;  front = front->link;
        delete p;
    }
}
template <class Type> void Queue<Type>::
```

```
EnQueue ( const Type & item ) {
    //append a new item into Queue
    //NULL Queue
    if ( front == NULL )
        front = rear = new QueueNode<Type>(item, NULL);
    //Not NULL
    else
        rear = →rearlink = new QueueNode<Type>(item,
           NULL);
}

template <class Type> Type Queue<Type>::
DeQueue ( ) {
    //Return the front item and then remove it from
       Queue
    assert ( !IsEmpty ( ) );
    QueueNode<Type> *p = front;
    Type retvalue = p->data;
    front = front->link;
    delete p;
    return retvalue;
}
```

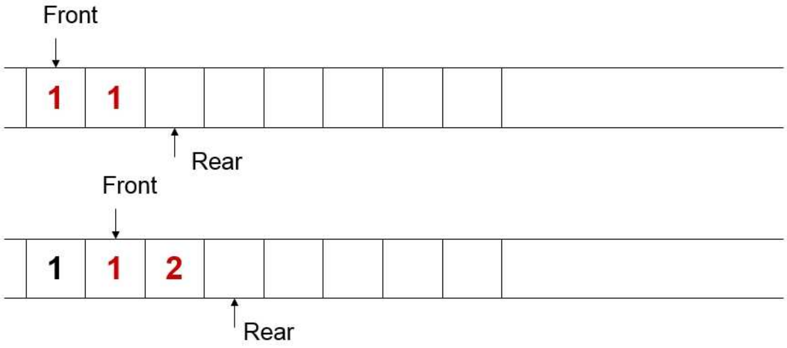
```



template <class Type> Type Queue<Type>::
GetFront ( ) {
    assert ( !IsEmpty ( ) );
    return front->data;
}
```

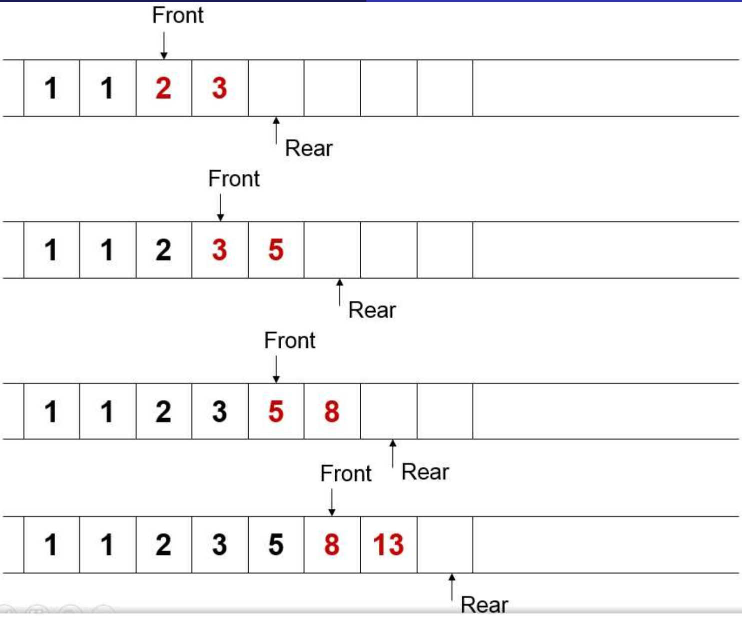
Next Subsection

Application 1: Fibonacci Array

$F_n=F_{n-1}+F_{n-2}$, $F_1=1$, $F_2=1$

1	1	2	3	5	8	13	21	34	...	...	...	...




Front
1
1
2
3
Rear
Front
1
1
2
3
5
Rear
Front
1
1
2
3
5
8
Rear
Front
1
1
2
3
5
8
13
Rear

Application 1: Fibonacci Array

```
#include "queue.h"
void Fibonacci ( int n ) {
    Queue <int> q; q.EnQueue (1); q.EnQueue (1);
    int s,t;
    s = q.DeQueue ();
    cout << s  << endl;
    for ( int i=2; i<=n; i++ ) {
         t = q.DeQueue ();
         q.EnQueue ( s+t );
         s = t;
         cout <<  s << endl;
      }
      printf ( “\n ” );
}
```

Application 2: Yangvi Triangle

Pascal's triangle

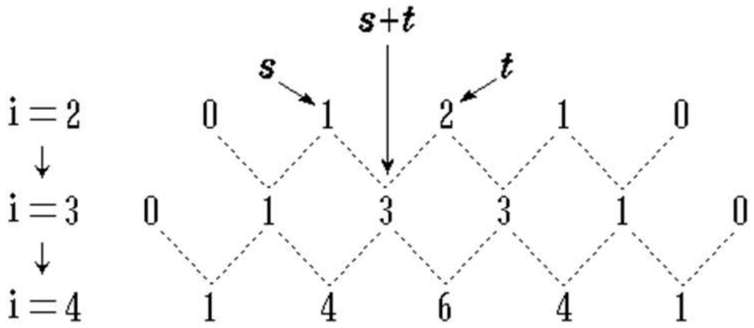
```
                        1                         i = 0
                     1     1                          1
                  1     2     1                       2
               1     3     3     1                    3
            1     4     6     4     1                 4
         1     5    10    10     5     1              5
      1     6    15    20    15     6     1           6
```

$$(a+b)^n = \sum_{i=0}^{n} C_n^i a^i b^{n-i}$$

$$= a^n + na^{n-1}b + \frac{1}{2}n(n+1)a^{n-2}b^2 + \cdots + nab^{n-1} + b^n$$

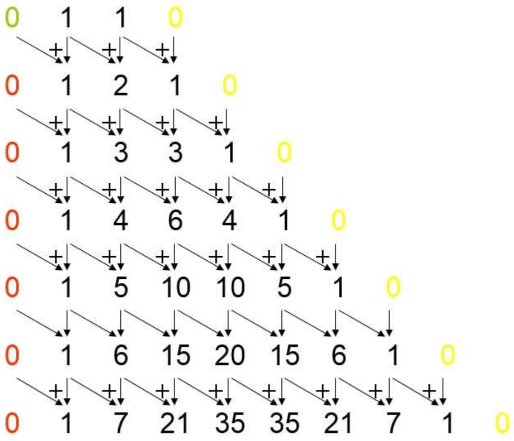
Application 2: Yangvi Triangle

Relation between i-th row and (i+1)-th elements



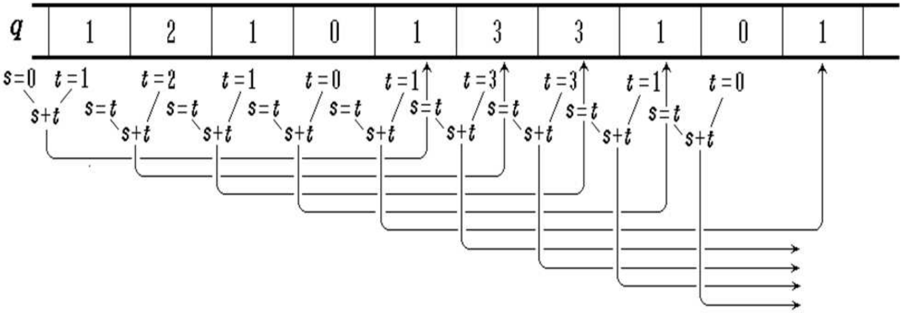
Application 2: Yangvi Triangle

Principle



Application 2: Yangvi Triangle

Principle



```
#include "queue.h"
void YANGVI ( int n ) {
    Queue <int> q; q.EnQueue(1);  q. EnQueue(1);
    int s = 0;
    for ( int i=1; i<=n; i++ ) {
      cout << endl;
      q. EnQueue(0);
      for ( int j=1; j<=i+2; j++ ) {
         int t;
         t=q.DeQueue();
         q.EnQueue( s+t );
         s = t;
         if ( j != i+2 ) cout << s << '' ;
      }
   }
}
```

Next Subsection

4.2.6 Priority Queue

- 优先级队列 是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素。
- 例如下表：任务的优先权及执行顺序的关系

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

优先队列的类定义

```
#include <assert.h>
#include <iostream.h>
#include <stdlib.h>
const int maxPQSize = 50; //缺省元素个数

template <class Type> class PQueue {
public:
    PQueue ( );
    ~PQueue ( ) { delete [ ] pqelements; }
    void PQInsert ( const Type & item );
    Type PQRemove ( );
    void makeEmpty ( ) { count = -1; }
    int IsEmpty ( ) const
           { return count == -1; }



    int IsFull ( ) const
```

```
            { return count == maxPQSize; }
    int Length ( ) const { return count; }
private:
    Type *pqelements; //存放数组
    int count; //队列元素计数
}
```

优先队列部分成员函数的实现

```
template <class Type>
PQueue<Type>::PQueue ( ) : count (-1) {
     pqelements = new Type[maxPQSize];
     assert ( pqelements != 0 ); //分配断言
}

template <class Type> void PQueue<Type> ::
PQInsert ( const Type & item ) {
     assert ( !IsFull ( ) ); //判队满断言
     count ; ++  pqelements[count] = item;
}





template <class Type>
Type PQueue<Type>::PQRemove ( ) {
    assert ( !IsEmpty ( ) );        //判队空断言
    Type min = pqelements[0];
    int minindex = 0;
    for (int i=1; i<count; i++) //寻找最小元素
       if ( pqelements[i] < min )   //存于min
       { min = pqelements[i];   minindex = i; }
    pqelements[minindex] = pqelements[count-1];
    count-- ;                    //删除
    return min;
}
```

Next Subsection

4.2.7 Summary

Next Subsection

4.2.8 Thinking...

- **Other special lists**
  - **Doubled-End Queue (双端队列)**
  - Double Stack (双栈)
  - Super Queue (超队列)
  - Super Stack (超栈)

双端队列是一种特殊的线性表，对它所有的插入和删除都限制在表的两端进行。这两个端点分别记作**end1**和**end2**。它好象一个特别的书架，取书和存书限定在两边进行。

Thinking . . .

- **Other special lists**
  - Doubled-End Queue (双端队列)
  - **Double Stack (双栈)**
  - Super Queue (超队列)
  - Super Stack (超栈)

双栈是一种加限制的双端队列，它规定从**end1**插入的元素只能从**end1** 端删除，而从**end2**插入的元素只能从**end2**端删除。它就好象两个底部相连的栈。

4.2.8 Thinking...

- **Other special lists**
  - Doubled-End Queue (双端队列)
  - Double Stack (双栈)
  - **Super Queue (超队列)**
  - Super Stack (超栈)

超队列是一种输出受限的双端队列，即删除限制在一端(例如**end1**)进行，而插入仍允许在两端进行。它好象一种特殊的队列，允许有的最新插入的元素最先删除。

4.2.8 Thinking...

- **Other special lists**
  - Doubled-End Queue (双端队列)
  - Double Stack (双栈)
  - Super Queue (超队列)
  - **Super Stack (超栈)**

超栈是一种输入受限的双端队列，即插入限制在一端**(例如end2)**进行，而删除仍允许在两端进行。它可以看成对栈溢出时的一种特殊的处理，即当栈溢出时，可将栈中保存最久**(end1** 端**)**的元素删除。