Data Structures and Algorithms (3)

Instructor: Ming Zhang

Textbook Authors: Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (the "Eleventh Five-Year" national planning textbook)

https://courses.edx.org/courses/PekingX/04830050x/2T2014/

Chapter 3 Stacks and Queues

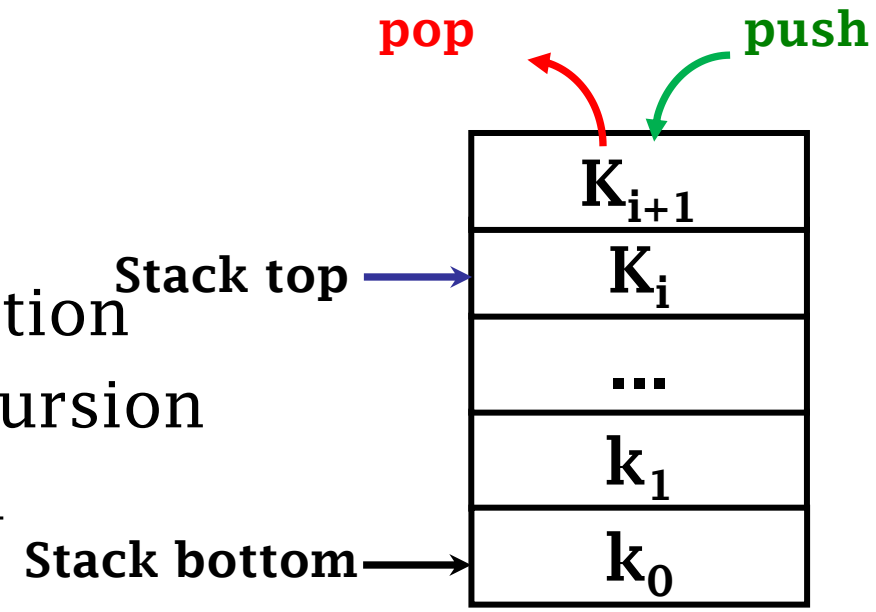
- Stacks
- Applications of stacks
  - Implementation of Recursion using Stacks
- Queues

Linear lists with limited operation

- **Stack**
  - Operation are permitted only on one end
- **Queue**
  - Operation are permitted only on two ends

Definition of stack

- **Last In First Out**
  - A linear list with limited access port
- **Main operation**
  - push、pop
- **Applications**
  - Expression evaluation
  - Elimination of recursion
  - Depth-first search

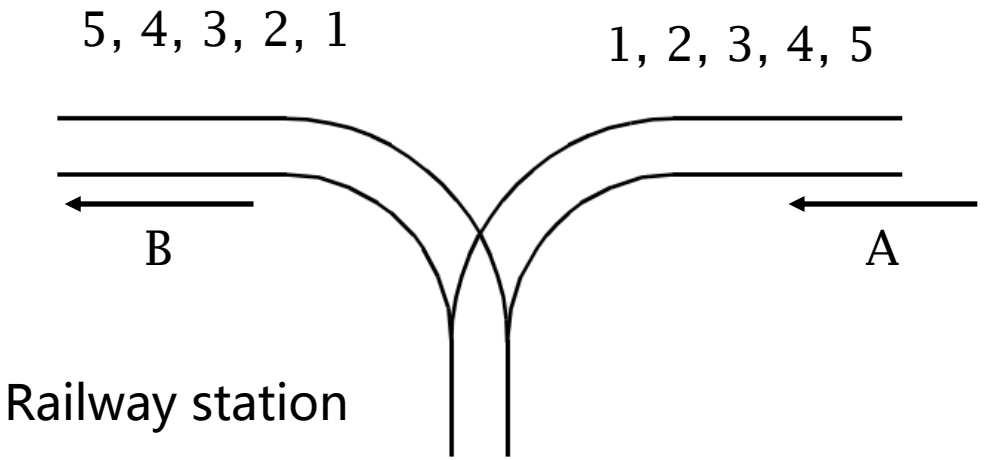


Abstract data type of stacks

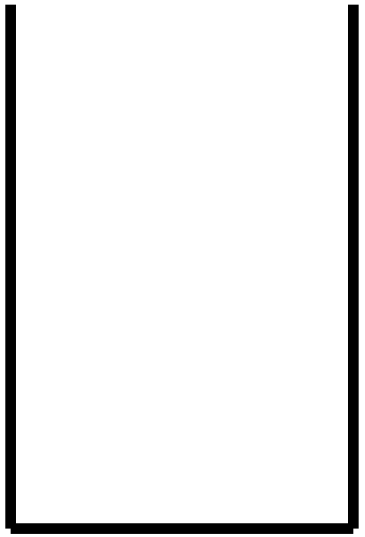
```
template <class T>
class Stack {
public:                    // Operation set of stacks
   void clear();           // Change into an empty stack
   bool push(const T item);
              // push item into the stack , return true if succeed, otherwise false
   bool pop(T& item);
             // pop item out of the stack ,  return true if succeed, otherwise false
   bool top(T& item);
             // read item at the top of the stack, return true if succeed, otherwise false
   bool isEmpty();      // If the stack is empty return true
   bool isFull();       // If the stack is full return true
};
```

Railway station problem

- Judge whether the trains go out of the station in the right order?
  - http://poj.org/problem?id=1363
- N trains numbered as 1,2,....,n go into the train in order , given an arrangement , judge whether the trains go out of the station in the right order?



Use legal reconstruction to find conflicts



Question

- If the order of the item pushed into the stack is 1,2,3,4 , then what is the order of the item popped out of the stack ?
- There is an original input sequence 1 , 2 , … , n , you are required to get the output sequence of p_1 , p_2 , … , p_n (They are a permutation of 1 , 2 , … , n)using a stack. If there exists subscript i , j , k , which meet the condition that i<j<k and $P_j<P_k<P_i$, then whether the output sequence is legal or not ?

Implementation of stacks

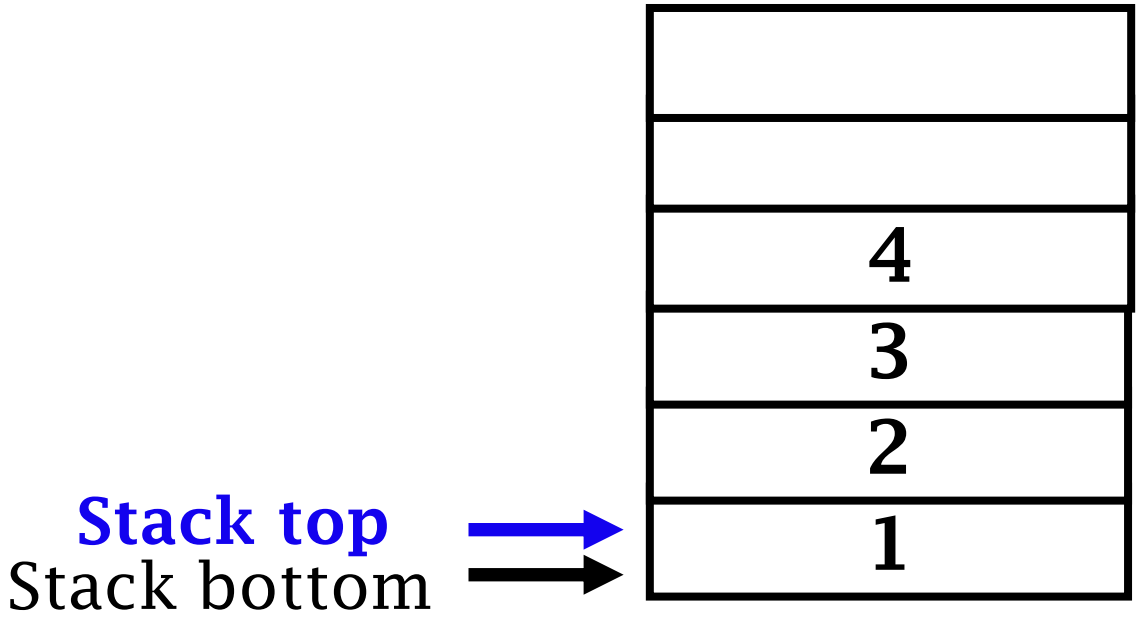
- **Array-based Stack**
  - Implemented by using vectors , is a simplified version of sequential list
    - The size of the stack
  - The key point is to make sure which end as the stack top
  - Overflow, underflow problem
- **Linked Stack**
  - Use single linked list for storage , in which the direction of the pointer is from stack top down

The class definition of Array-based Stack

```
template <class T> class arrStack : public Stack <T> {
private:                  // storage of Array-based Stack
   int mSize;            // the number of elements that the stack can have at most
   int top;               // stack top , should be small than mSize
   T *st;                // array to put stack element
public:                   // implementation of the operation of the Array-based Stack
   arrStack(int size) {
      // creates an instance of Array-based Stack with given size
      mSize = size; top = -1; st = new T[mSize];
   }
   arrStack() {// creates an instance of Array-based Stack
      top = -1;
   }
   ~arrStack() { delete [] st; }
   void clear() { top = -1; }  // clear the stack
}
```

Array-based Stack

- The index of the last element pushed into the stack is 4 , followed by 3,2,1 in order



Overflow of Array-based Stack

- **Overflow**
  - When you perform push operation on a full stack (that already has maxsize elements), overflow will occur.
- **Underflow**
  - When you perform pop operation on an empty stack, underflow will occur.

Push

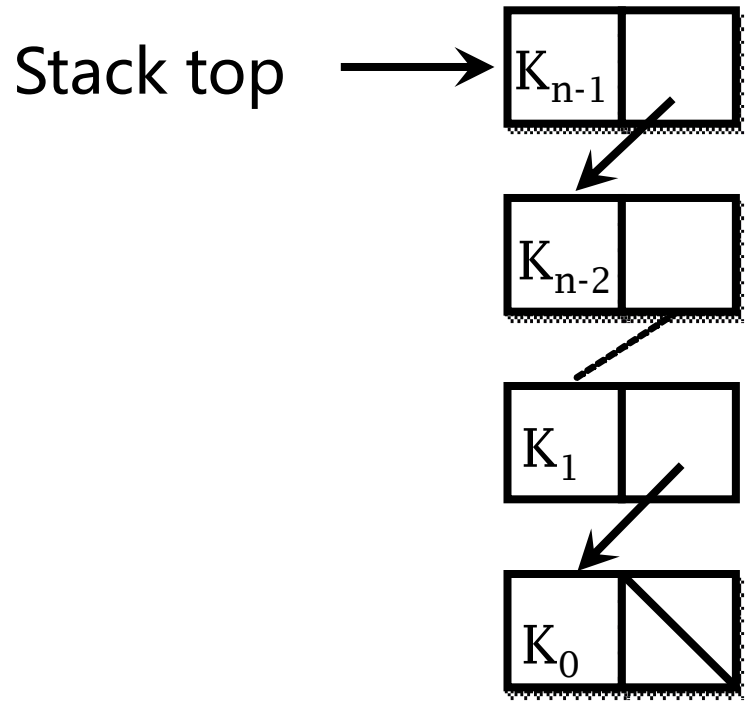
```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) {
// the stack has been full
        cout <<  “Stack overflow" << endl;
        return false;
    } else { //push new element into the stack and
modify the pointer of the stack top
        st[++top] = item;
        return true;
    }
}
```

Pop

```
bool arrStack<T>::pop(T & item) {  // pop
    if (top == -1) {      // the stack is empty
        cout << " The stack is empty, you can’t
pop "<< endl;
        return false;
    } else {
        // Get top value and decrease top by 1
        item = st[top--];
        return true;
    }
}
```

Definition of Linked Stack

- Use single linked list for storage
- The direction of the pointer is from stack top down



Construction of Linked Stack

```
template <class T> class lnkStack : public Stack <T> {
private:                    // storage for linked stack
    Link<T>* top;
    //Pointer which points to the stack top
    int size; // the number of elements that the stack can
have at most
public:// implementation of the operation of the linked Stack
    lnkStack(int defSize) {  // constructed function
        top = NULL; size = 0;
    }
    ~lnkStack() {           // destructor function
        clear();
    }
}
```

Push

```
// implementation of push operation of linked stack
bool lnksStack<T>:: push(const T item) {
    Link<T>* tmp = new Link<T>(item, top);
    top = tmp;
    size++;
    return true;
}

Link(const T info, Link* nextValue)  {
        // constructed function with 2 parameters
        data = info;
        next = nextValue;
}
```

Pop

```
// implementation of pop operation of linked stack
bool lnkStack<T>:: pop(T& item) {
    Link <T> *tmp;
    if (size == 0) {
        cout << " The stack is empty, you can't pop"<< endl;
        return false;
    }
    item = top->data;
    tmp = top->next;
    delete top;
    top = tmp;
    size--;
    return true;
}
```

Comparison of Array-based Stack and Linked Stack

- **Time efficiency**
  - All operations only take constant time
  - Array-based Stack and Linked Stack have almost the same time efficiency
- **Space efficiency**
  - The length of an Array-based Stack is fixed
  - The length of a Linked Stack is variable, with extra structural cost

Comparison of Array-based Stack and Linked Stack

- In real applications , Array-based Stack is more widely used than Linked Stack
  - It is easy for Array-based Stack to perform relative replacement according to the position of stack top , quickly position and read the internal element
  - The time taken for Array-based Stack to read internal element is O(1). And the Linked stack has to walk along the chain of pointers, and is slower than Array-based Stack . It takes O(k) time to read the kth element.
- In general, the stack does not allow the internal operation, can only operate in the stack top

Question : functions about stack in STL

- Top function gets the element of the stack top and returns the result back to the user
- Pop function pops a element out of the stack top (if the stack is not empty)
  - Pop function is just an operation and doesn't return the result
  - pointer = aStack.pop() ? Error !
- Why does STL divide these two operations ? Why not provide ptop ?

Applications of stacks

- Characteristic of stacks : **last-in first-out**
  - Embodies the transparency between elements
- Commonly used to deal with data which has recursive structure
  - **DFSevaluate the expression**
  - Subroutine / function call management
  - **Elimination of recursion**

Evaluate the expression

- Recursive definition of expressions
  - The basic symbol set : {0 , 1 , ... , 9 , + , - , * , / , (,) }
  - Grammar set : {<expression> , <term> , <factor> , <constant> , <digit> }
- The infix expression 23+(34*45)/(5+6+7)
- Postfix expression 23 34 45 * 5 6 + 7 + / +

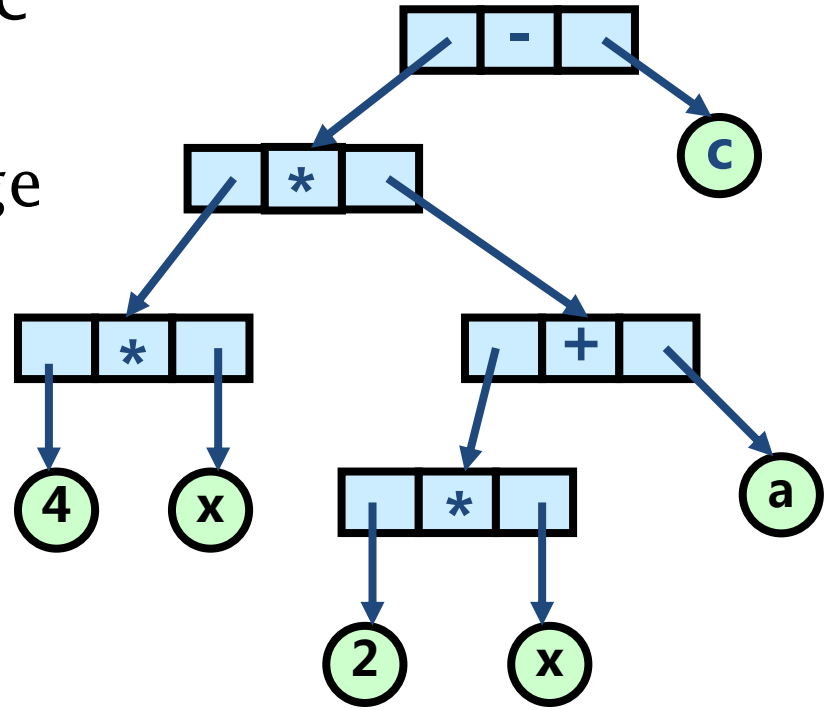
Infix expression

- **Infix expression**

  4 * x * (2 * x + a) – c

  – Operator in the middle

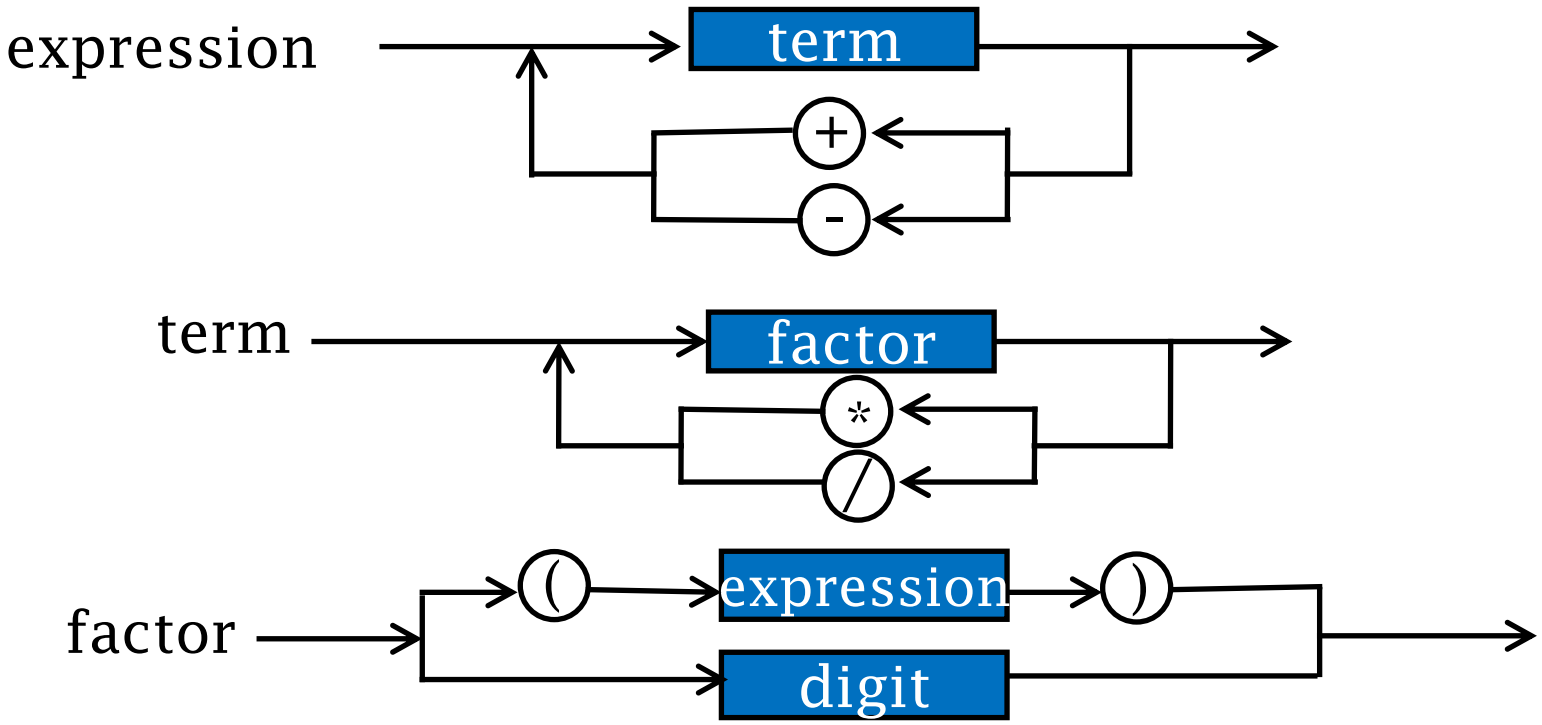
  – Need brackets to change the priority



Syntax formula for infix expression

<expression> ::= <term> <term> +
| <term> <term> -
| <term>
<term> ::= <factor> < factor > *
| < factor > < factor > /
| < factor >
< factor > ::= < constant >
< constant > ::= <digit>
| <digit> <constant>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Graphical representation for expression recursion

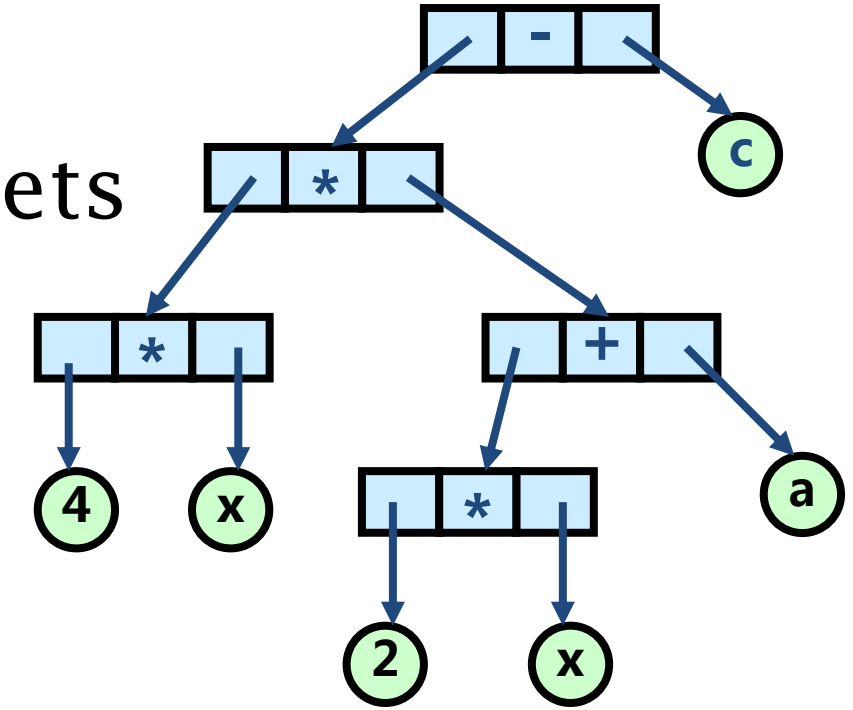


Postfix expression

- **Postfix expression**

  4 x * 2 x * a + * c -

  - Operators behind
  - No need for brackets



Postfix expression

<expression> ::= <term> <term> +
| <term> <term> -
| <term>
<term> ::= <factor> < factor > *
| < factor > < factor > /
| < factor >
< factor > ::= < constant >
< constant > ::= <digit>
| <digit> <constant>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Evaluating a postfix expression

- 23 34 45 * 5 6 + 7 + / + = ?

Calculation characteristics ?

The main differences between infix and postfix expression ?

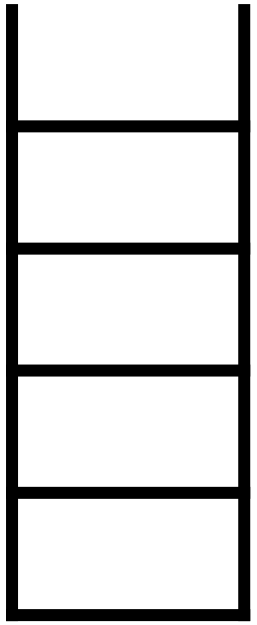
23 + 34 * 45 / (5 + 6 + 7) = ?

23 34 45 * 5 6 + 7 + / + = ?

postfix expression to be handled :

23 34 45 * 5 6 + 7 + / +

change of the stack state



	1180

calculation result

Evaluating a postfix expression

- Loop : read symbol sequences of expressions (assume "=" as the end of the input sequence) , and analyze one by one according to the element symbol read
  1. When an operand is met , push
  2. When an operator is met, pop twice and get two operands, calculate them using the operator. And finally push the result into the stack.
- Continue the process above until the symbol " = " is met , then the value of the stack top is the value of the input expression

The class definition of postfix calculator

```
class Calculator {
private:
    Stack<double> s;//the stack is used for pushing and storing operands
    // push two operands opd1 and opd2 from the stack top
    bool GetTwoOperands(double& opd1,double& opd2);
    // get two operands, and calculate according to op
    void Compute(char op);
public:
    Calculator(void){} ;
    // creates calculator instance and construct a new stack
    void Run(void);   // read the postfix expression, ends when meet "="
    void Clear(void); // clear the calculator to prepare for the next calculation
};
```

The class definition of postfix calculator

```
template <class ELEM>
bool Calculator<ELEM>::GetTwoOperands(ELEM& opnd1, ELEM& opnd2) {
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd1 = S.Pop(); // right operator
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd2 = S.Pop(); // left operator
    return true;
}
```

The class definition of postfix calculator

```
template <class ELEM> void Calculator<ELEM>::Compute(char op) {
    bool result;  ELEM operand1, operand2;
    result = GetTwoOperands(operand1, operand2);
    if (result == true)
        switch(op) {
            case '+' : S.Push(operand2 + operand1); break;
            case '-' : S.Push(operand2 - operand1); break;
            case '*' : S.Push(operand2 * operand1); break;
            case '/' : if (operand1 == 0.0) {
                    cerr << "Divide by 0!" << endl;
                    S.ClearStack();
                } else S.Push(operand2 / operand1);
                break;
        }
    else S.ClearStack();
}
```

The class definition of postfix calculator

```
template <class ELEM> void Calculator<ELEM>::Run(void) {
    char c; ELEM newoperand;
    while (cin >> c, c != '=') {
        switch(c) {
            case '+': case '-': case '*': case '/':
                Compute(c);
                break;
            default:
                cin.putback(c); cin >> newoperand;
                Enter(newoperand);
                break;
        }
    }
    if (!S.IsEmpty())
        cout << S.Pop() << endl;        // print the final result
}
```

Question

- 1. Stack is usually implemented by using single linked list. Can we use doubly linked list? Which is better ?

- 2. Please summarize the properties of prefix expression, as well as the evaluation process.

Chapter 3 Stacks and Queues

- Stacks
- **Application of stacks**
  - **Implementation of Recursion using Stacks**
- Queues

Transformation from recursion to non-recursion

- The principle of recursive function
- Transformation of recursion
- The non recursive function after optimization

The principle of recursive function

Another study of recursion

- Factorial $f(n) = \begin{cases} n \times f(n-1) & n \geq 1 \\ 1 & n = 0 \end{cases}$
- **Exit of recursion**
  - End condition of recursion is when the minimal problem is solved
  - More than one exits are permitted
- **Rule of recursion**

  (Recursive body + bounded function)
  - Divide the original problem into sub problems
  - Ensure that the scale of recursion is more and more closer to the end condition

The principle of recursive function

Non recursive implementation of recursive algorithm

$$f(n) = \begin{cases} n \times f(n-1) & n \geq 1 \\ 1 & n = 0 \end{cases}$$

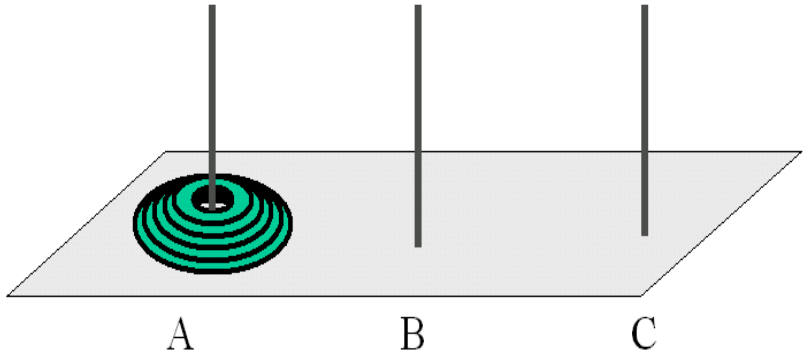
- Non recursive implementation of factorial
  - Establish iteration
  - Transformation from recursion to non-recursion

- How about the problem of Hanoi Tower ?

The principle of recursive function

Recursion program for Hanoi tower problem

http://www.17yy.com/f/play/89425.html

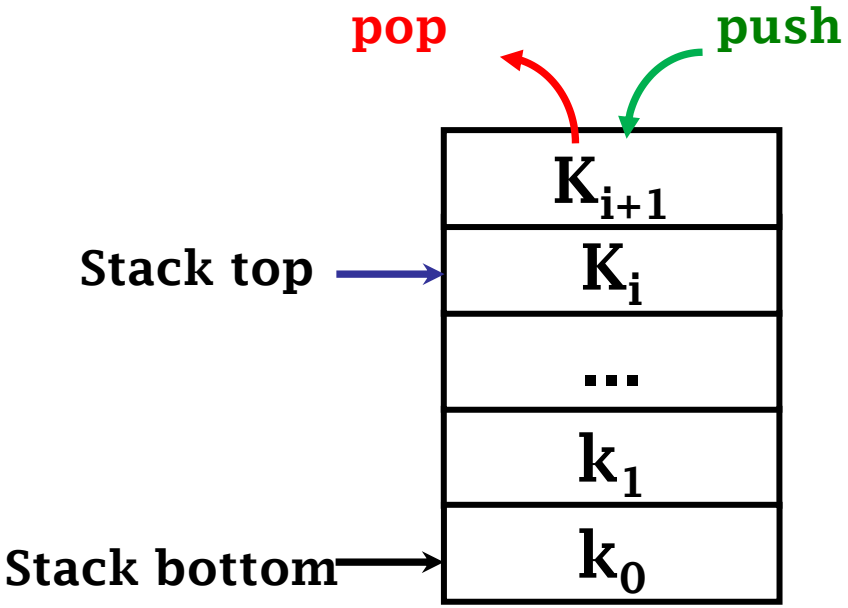


- hanoi(n,X,Y,Z)
  - Move n disk
  - Move the disk from pillar X to pillar Z
  - X、Y、Z can be used to place disks temporarily
    - Big disks cannot be put on small disks
- Such as hanoi(2, 'B', 'C' , 'A')
  - Move 2 disks from pillar B to pillar A

3.1.3 Transformation from recursion to non-recursion

```
void hanoi(int n, char X, char Y, char Z) {
    if (n <= 1)
        move(X,Z);
    else  {
// don’t move the largest disk on X and move the left n-1 disk to Y
        hanoi(n-1,X,Z,Y);
        move(X,Z);        //move the largest disk on X to Z
        hanoi(n-1,Y,X,Z); // move the n-1 disk on Y to Z
    }
}
void move(char X, char Y)
// move the disk on the top of pillar x to pillar Y
{
  cout  << "move"  <<  X  << "to" << Y  << endl;
}
```

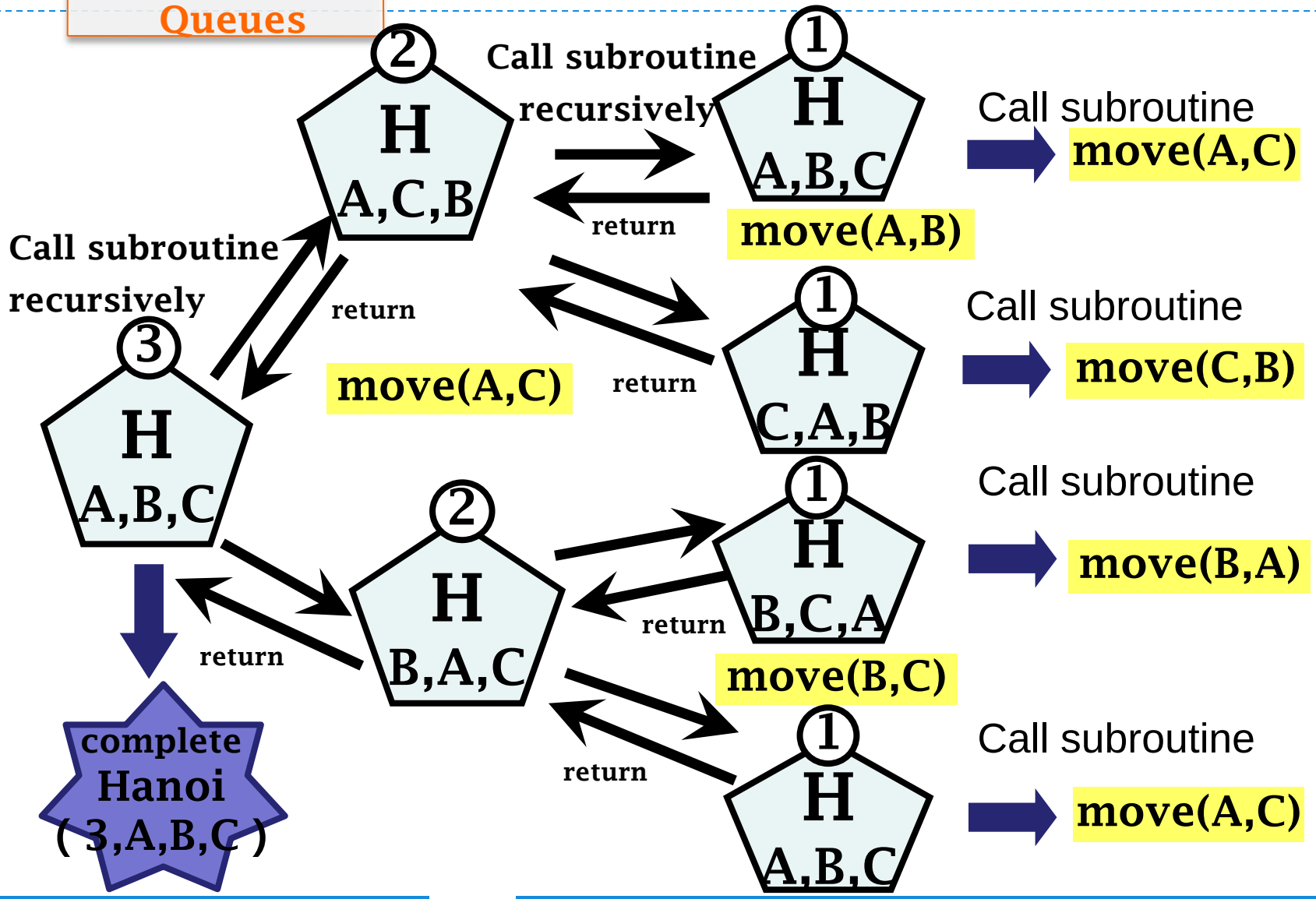
Operating diagram of Hanoi recursive subroutine

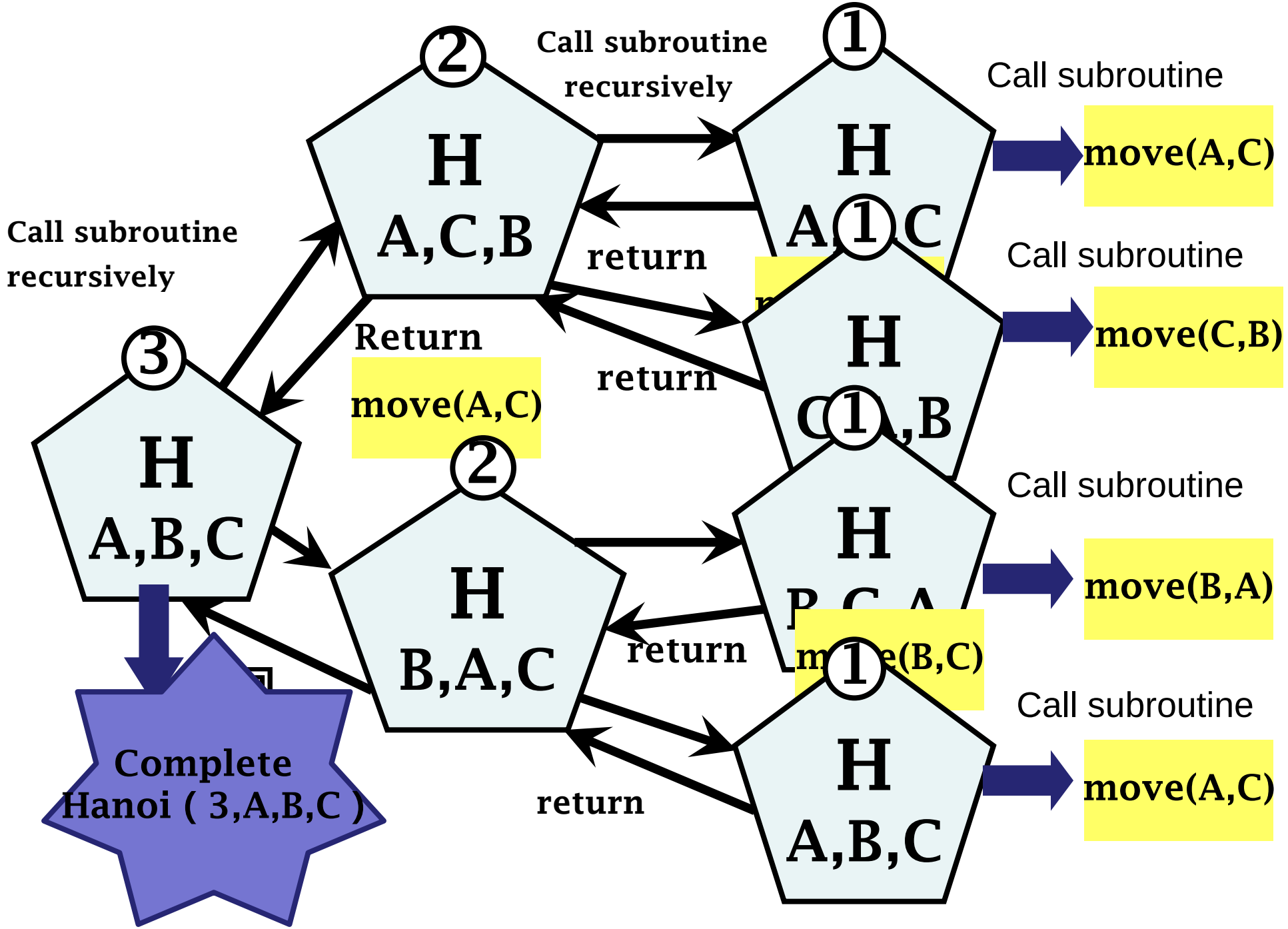


Execute the instructions of Hanoi program

Exchange information with subroutine via stack

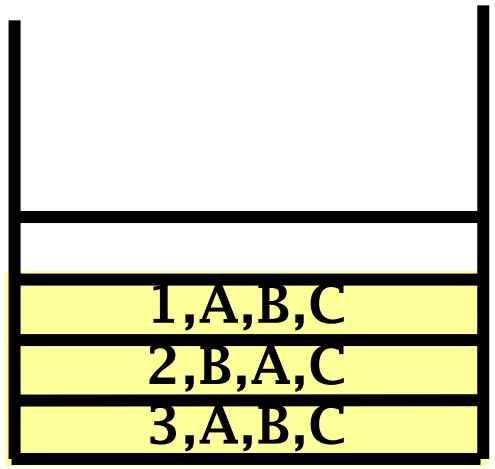
3.1.3 Transformation from recursion to non-recursion




Complete
Hanoi (3,A,B,C)
3
H
A,B,C
Call subroutine
recursively
2
H
A,C,B
Call subroutine
recursively
1
H
A,1,C
Call subroutine
move(A,C)
return
Return
move(A,C)
H
C,1,B
Call subroutine
move(C,B)
return
2
H
B,A,C
H
B,C,A
Call subroutine
move(B,A)
return
1
H
A,B,C
Call subroutine
move(A,C)
return

The status of stack when the recursion is executed

hanoi(1,A,B,C)
hanoi(1,B,C,A)
hanoi(2,B,A,C)
hanoi(1,C,A,B)
hanoi(1,A,B,C)
hanoi(2,A,C,B)
hanoi(3,A,B,C)



Perform move(A,C)

A recursive mathematical formula

$$fu(n) = \begin{cases} n+1 & when \quad n<2 \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \end{cases}$$

Example for recursive function

```
int f(int n) {
  if (n<2)
      return n+1;
  else
      return f(n/2) * f(n/4);
}
```

$$fu(n) = \begin{cases} n+1 & when \quad n<2 \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \end{cases}$$

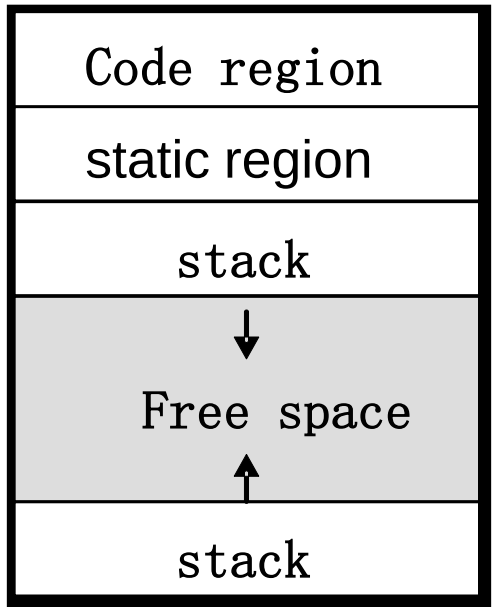
Example for recursive function(change a little)

```
void exmp(int n, int& f) {
    int u1, u2;
    if (n<2)
        f = n+1;
    else {
        exmp((int)(n/2), u1);
        exmp((int)(n/4), u2);
        f = u1*u2;
    }
}
```

$$fu(n) = \begin{cases} n+1 & when \quad n<2 \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \end{cases}$$

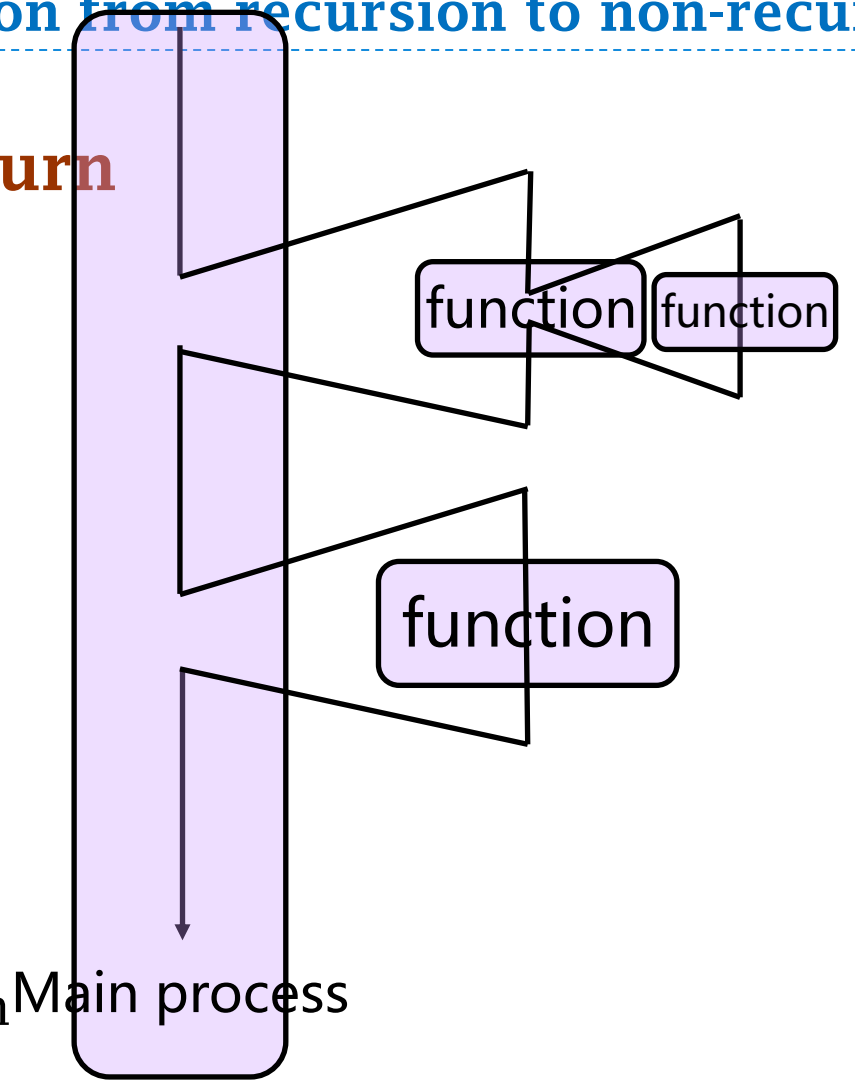
Dynamic memory allocation when the function is executed

- **Stack** is used for data that match last-in and first-out after allocated
  - Such as call function
- **Heap** is used for data which doesn't match LIFO
  - Such as the distribution of the space that the pointer points to



Function call and the steps of return

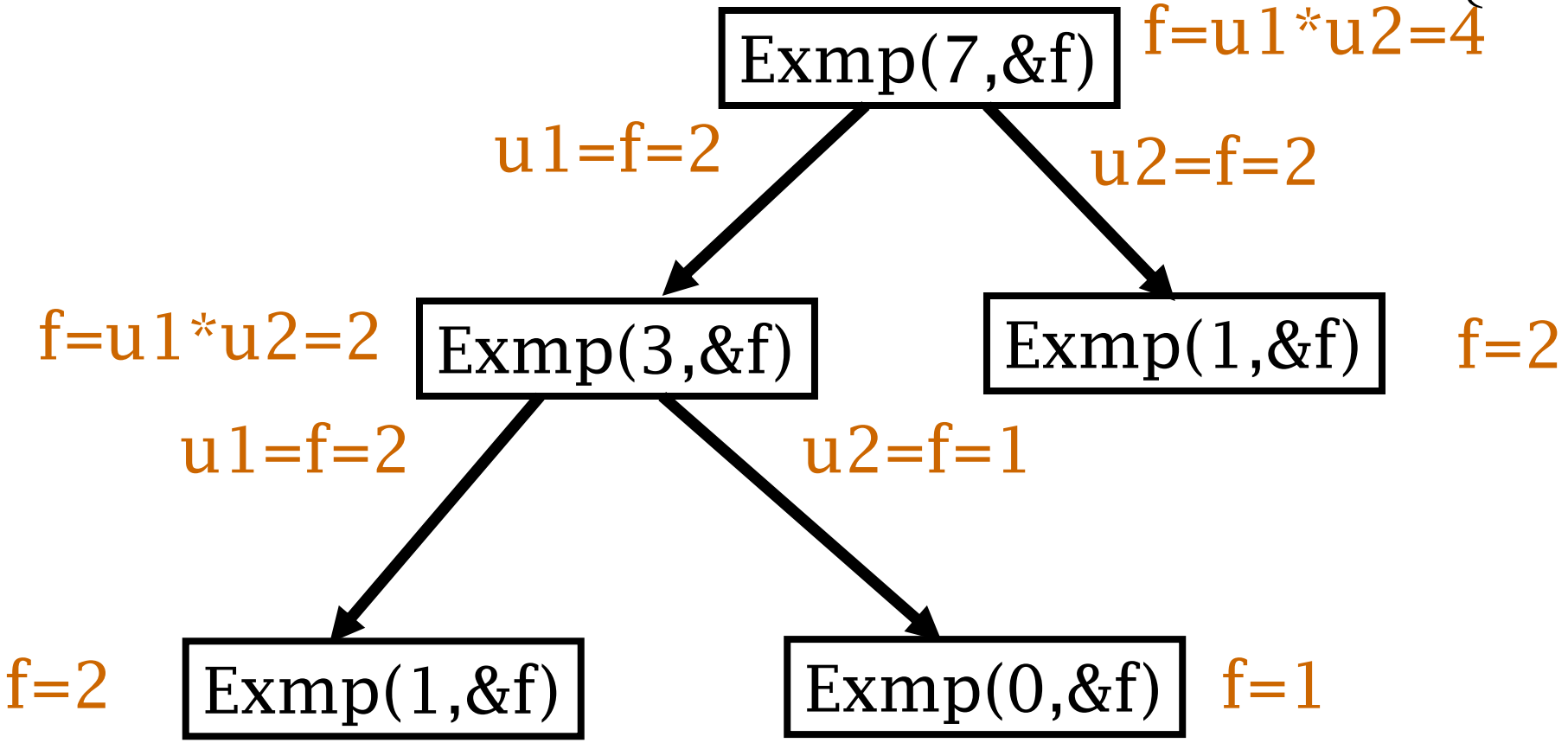
- **Function recall**
  - Save call information (parameter , return address)
  - Distribute data area (Local variable)
  - Control transfers to the exit of the function called
- **Return**
  - Save return information
  - Release data area
  - Control transfers to a superior function (the main call function)



3.1.3 Transformation from recursion to non-recursion

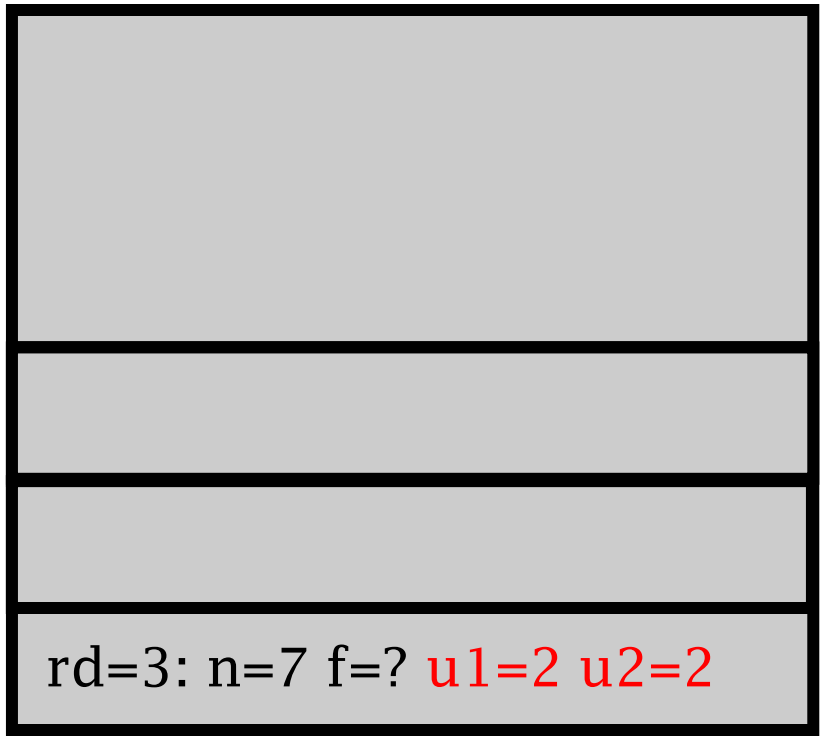
Diagram for the process of executing function

$$fu(n)=\begin{cases} n+1 & when\ \ n<2 \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2 \end{cases}$$



Simulate the process of recursion call by stack

- Last call , first return (LIFO) , so stack is used




```
void exmp(int n, int& f) {
    int u1, u2;
    if (n<2) f = n+1;
    else {
        exmp((int)(n/2), u1);
        exmp((int)(n/4), u2);
        f = u1*u2;
    }
}
```

Question

- For following function , please draw the recursive tree when n=4 case, and use stack to simulate the process of recursive calls with the stack
  - The factorial function

    $f_0=1$, $f_1=1$, $f_n = n\ f_{n-1}$
  - 2 order Fibonacci function

    $f_0=0$, $f_1=1$, $f_n = f_{n-1}+ f_{n-2}$

Chapter 3 Stacks and Queues

- Stacks
- **Applications of stacks**
  - **Implementation of Recursion using Stacks**
- Queues

Transformation from recursion to non-recursion

- The principle of recursive function
- **Transformation of recursion**
- **The non recursive function after optimization**

(2) Transformation of recursion

Method of transform recursion to non-recursion

$$fu(n) = \begin{cases} n+1 & when \quad n<2 \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \end{cases}$$

- Direct transformation method

1.Set a working stack to record the current working record
2. Set t+2 statement label
3. Increase non recursive entrance
4. Replace the i-th (i = 1, …, t)recursion rule
5. Add statement : "goto label t+1" at all the Recursive entrance
6. The format of the statement labeled t+1
7. Rewrite the recursion in circulation and nest
8. Optimization

rd=2: n=0 f=? u1=? u2=?
rd=1: n=3 f=? u1=2 u2=?
rd=3: n=7 f=? u1=? u2=?

(2) Transformation of recursion

1. Set a working stack to record the working record

- All the parameters and local variables that occur in the function must be replaced by the corresponding data members in the stack
  - Return statement label domain (t+2 value)
  - Parameter of the function(parameter value, reference type)
  - Local variable

```
typedef struct elem { // ADT of  stacks
   int rd;              // return the label of the statement
   Datatypeofp1 p1;           // parameter of the function
   …
   Datatypeofpm pm;
   Datatypeofq1 q1;           // local variable
   …
   Datatypeofqn qn;
} ELEM;
```

2. Set t+2 statement label

- label *0* : The first executable statement
- label t+1 : set at the end of the function body
- label i (1<=i<=t) : the ith return place of the recursion

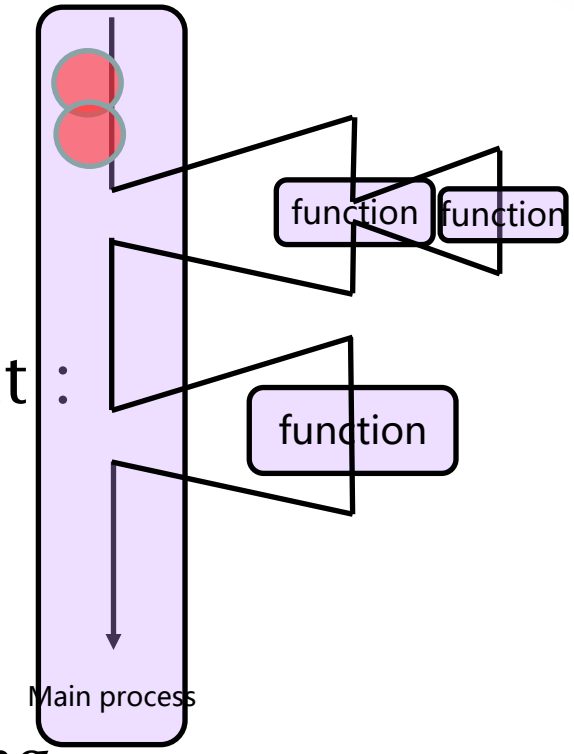
3. Increase non recursive entrance

// push

S.push(t+1 , p1, … , pm , q1 , …qn);

(2) Transformation of recursion

4. Replace the ith (i = 1, …, t)recursion rule



- Suppose the ith (i=1, …, t) recursive call statement is : recf(a1, a2, …,am) ;
- Then replace it with the following statement :

```
S.push(i, a1, …, am) ;  // Push the actual parameter
goto label 0 ;
.....
label i :  x = S.top() ; S.pop();
```

/* pop , and assign some value of X to the working record of stack top S.top()— It is equivalent to send the value of reference type parameter back to the local variable*/

(2) Transformation of recursion

5. Add statement at all the Recursive entrance

- goto label *t+1*;

6. The format of the statement labeled t+1

```
switch ((x=S.top ()).rd) {
    case 0 :    goto label 0;
                break;
    case 1 : goto label 1;
                    break;
    ..........
    case t+1 : item = S.top(); S.pop(); // return
               break;
    default : break;
}
```

7. Rewrite the recursion in circulation and nest

- For recursion in circulation , you can rewrite it into circulation of goto type which is equivalent to it
- For nested recursion call

  For example , recf (... recf())

  Change it into :

  $exmp_1$ = recf ( );

  $exmp_2$ = recf ($exmp_1$);

  ...

  $exmp_k$ = recf ($exmp_{k-1}$)

  Then solve it use the rule 4

8. Optimization

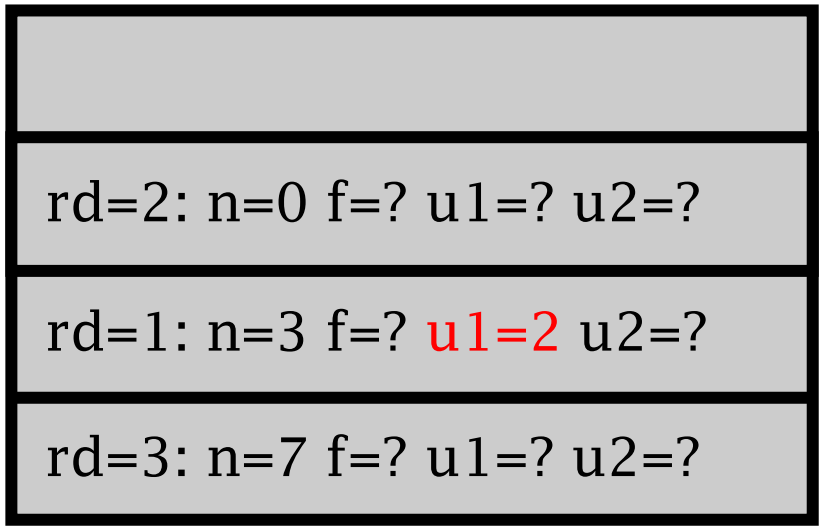
- Further optimization
  - Remove redundant push and pop operation
  - According to the flow chart to find the corresponding cyclic structure, there by eliminating the goto statement

(2) Transformation of recursion

Definition of data structure

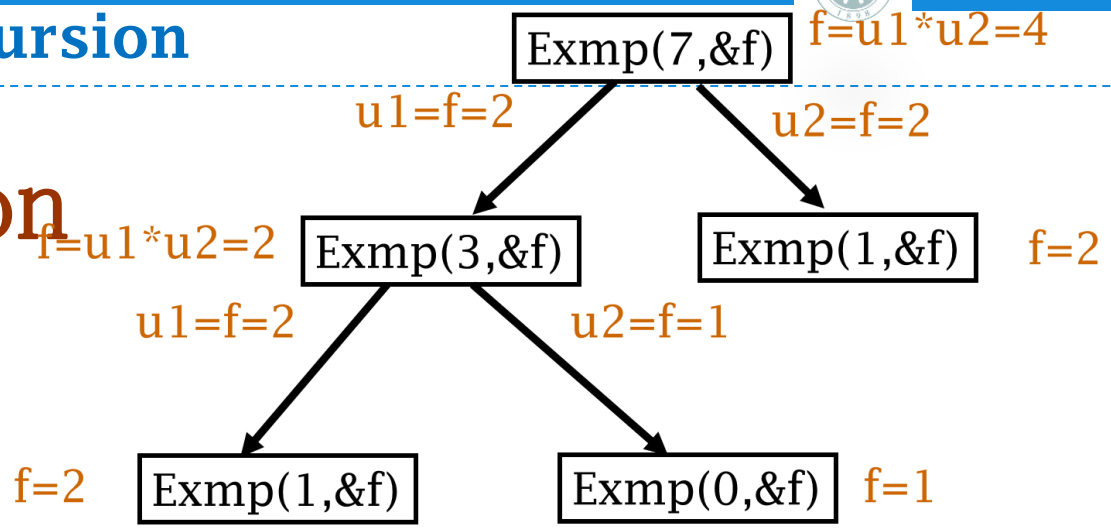
$$fu(n) = \begin{cases} n+1 & when \quad n<2 \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \end{cases}$$

```
typedef struct elem {
    int rd, pn, pf, q1, q2;
} ELEM;
class nonrec {
    private:
        stack <ELEM> S;
    public:
        nonrec(void) { }      // constructor
        void replace1(int n, int& f);
};
```



(2) Transformation of recursion

Entrance of recursion



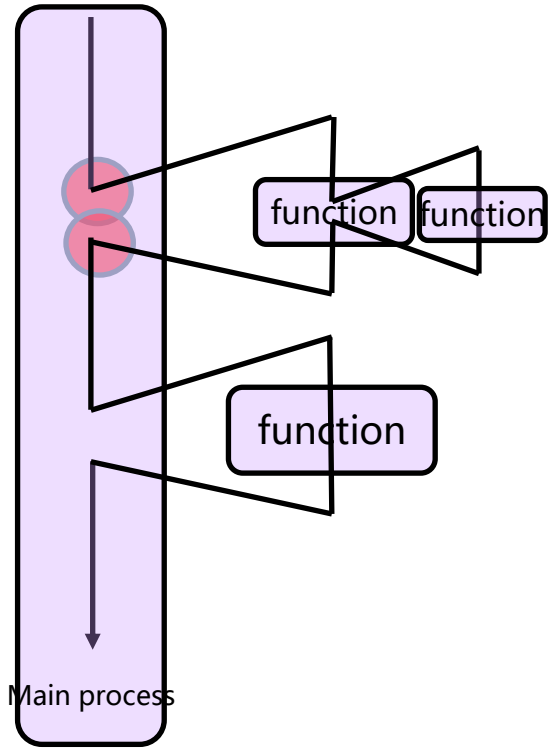

```
void nonrec::replace1(int n, int& f)  {
   ELEM x, tmp
   x.rd = 3;    x.pn = n;
   S.push(x);    // pushed into the stack bottom as lookout
label0: if ((x = S.top()).pn < 2)  {
          S.pop( );
          x.pf = x.pn + 1;
          S.push(x);
          goto label3;
        }
```

(2) Transformation of recursion

The first recursion statement

$$fu(n) = \begin{cases} n+1 & when \quad n<2 \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \end{cases}$$

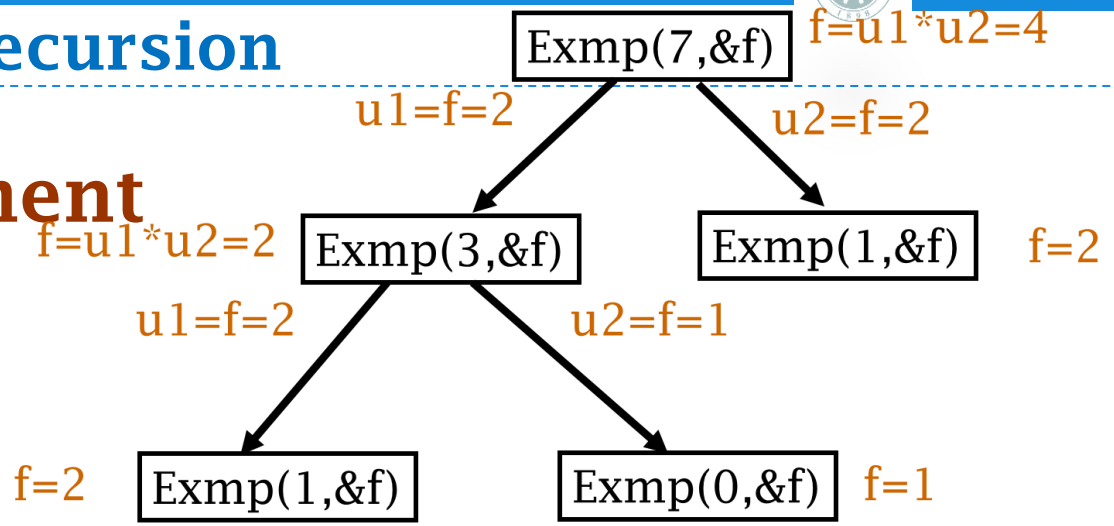
```
x.rd = 1;      // the first recursion
x.pn = (int)(x.pn/2);
 S.push(x);
goto label0;
label1: tmp = S.top(); S.pop();
x = S.top(); S.pop();
x.q1 = tmp.pf; // modify u1=pf
S.push(x);
```



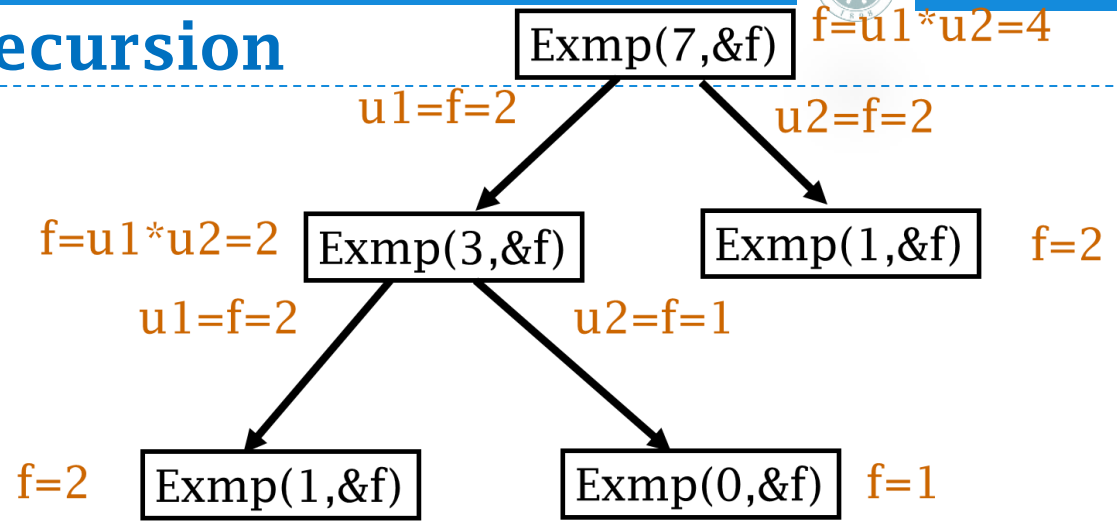
(2) Transformation of recursion

The second recursion statement

```
        x.pn = (int)(x.pn/4);
        x.rd = 2;
        S.push(x);
        goto label0;
label2: tmp = S.top(); S.pop();
        x = S.top(); S.pop();
        x.q2 = tmp.pf;
        x.pf = x.q1 * x.q2;
        S.push(x);
```

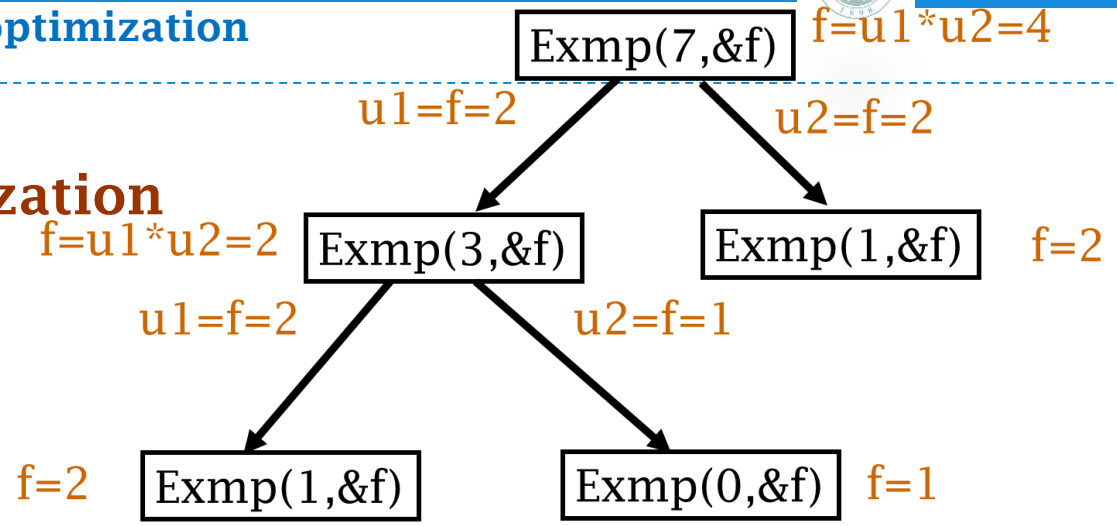


(2) Transformation of recursion



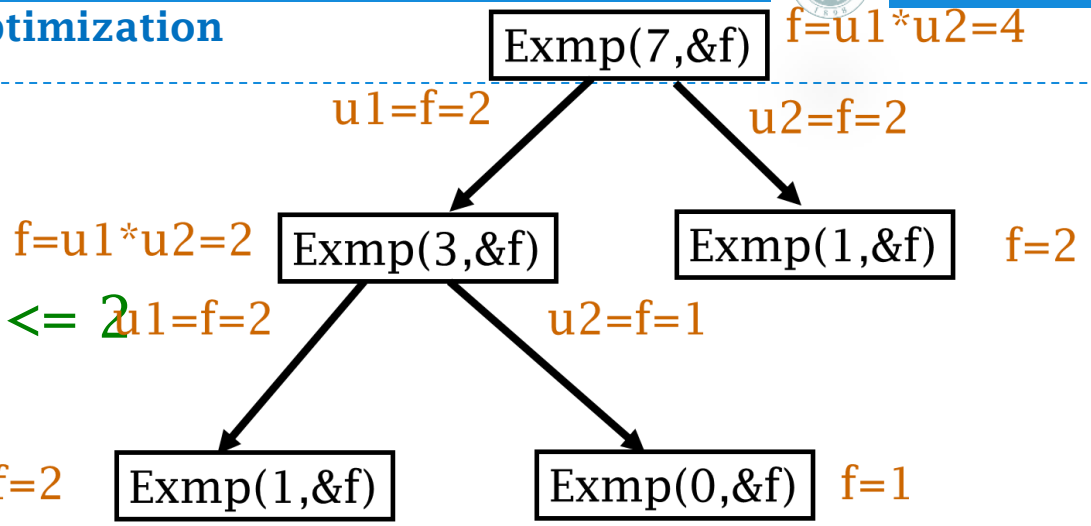

```
label3: x = S.top();
      switch(x.rd)  {
         case 1 : goto label1;
                  break;
         case 2 : goto label2;
                  break;
         case 3 : tmp = S.top(); S.pop();
                  f = tmp.pf;          //finish calculating
                  break;
         default : cerr << "error label number in stack";
                   break;
       }
}
```

The non recursive function after optimization



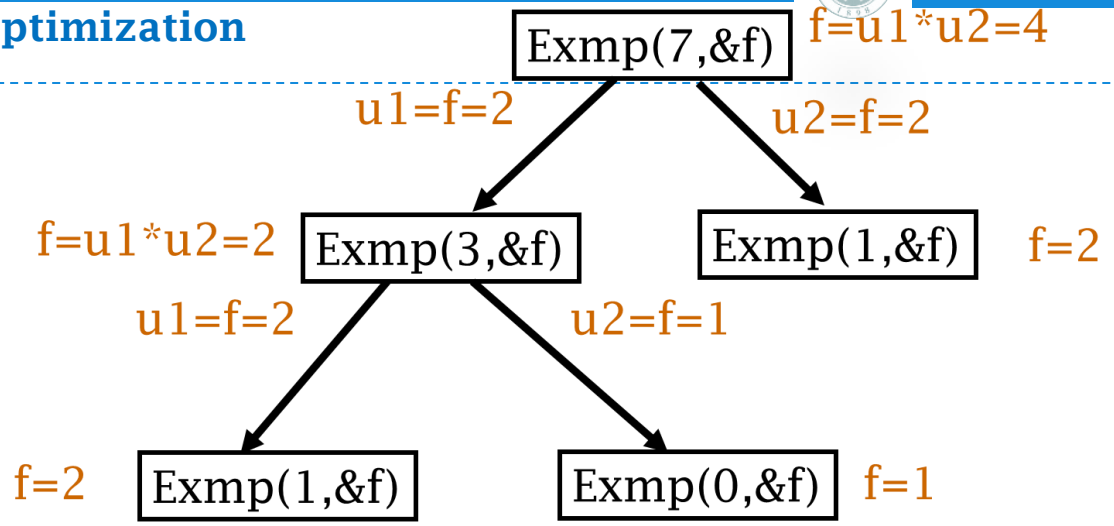

```
void nonrec::replace2(int n, int& f)  {
    ELEM x, tmp;
     // information of the entrance
    x.rd = 3;   x.pn = n;   S.push(x);
    do  {
      // go into the stack along the left side
       while ((x=S.top()).pn >= 2){
          x.rd = 1;
          x.pn = (int)(x.pn/2);
          S.push(x);
       }
```

(3) The non recursive function after optimization




```
        x = S.top(); S.pop();   // initial entrance , n <= 2
        x.pf = x.pn + 1;
        S.push(x);
        // If it is returned from the second recursion
then rise
        while ((x = S.top()).rd==2) {
          tmp = S.top(); S.pop();
          x = S.top(); S.pop();
          x.pf = x.q * tmp.pf;
          S.push(x);
        }
```

(3) The non recursive function after optimization




```
        if ((x = S.topValue()).rd == 1) {
            tmp = S.top(); S.pop();
            x = S.top(); S.pop();
            x.q = tmp.pf;    S.push(x);
            tmp.rd = 2;  // enter the second recursion
            tmp.pn = (int)(x.pn/4);
            S.push(tmp);
        }
    } while ((x = S.top()).rd != 3);
    x = S.top(); S.pop();
    f = x.pf;
}
```

Performance experiment of transformation from recursion to non recursive

Comparison of quicksort (unit ms)

Method \ Scale	10000	100000	1000000	10000000
Quicksort with recursion	4.5	29.8	268.7	2946.7
Quicksort with non recursive fixed method	1.6	23.3	251.7	2786.1
Quicksort with non recursive unfixed method	1.6	20.2	248.5	2721.9
Sort in STL	4.8	59.5	629.8	7664.1

Note : testing environment
Intel Core Duo CPU T2350
Memory 512MB
Operating system Windows XP SP2
Programming environment Visual C++ 6.0

Performance experiment of transformation from recursion to non recursive

Scale of processing problems using recursion and non recursive method

- Evaluate f(x) by recursion:

```
int f(int x) {
    if (x==0) return 0;
    return f(x-1)+1;
}
```

  - Under the default settings, when x exceed **11,772** ,the stack overflow may occur.
- Evaluate f(x) by non recursive method，the element in the stack record the current x and the return value
  - Under the default settings, when x exceed **32,375,567** , error may occur

Questions

- Use the direct transformation for …
  - The factorial function
  - 2-order Fibonacci function
  - Hanoi Tower algorithm

Chapter 3 Stacks and Queues

- Stacks
- Applications of Stacks
  - Implementation of Recursion using Stacks
- **Queues**

Definition of queues

- **First In First Out**
  - Linear lists that limit accessing point
    - Release elements according to the order of arrival
    - All the insertions occur at one end of the list and all the deletions occur at the other end
- **Main elements**
  - front
  - rear

Main operations of queues

- Insert an element into the queue (enQueue)
- Remove an element from the queue (deQueue)
- Get the element in the front (getFront)
- Judge whether the queue is empty (isEmpty)

Abstract data type of queues

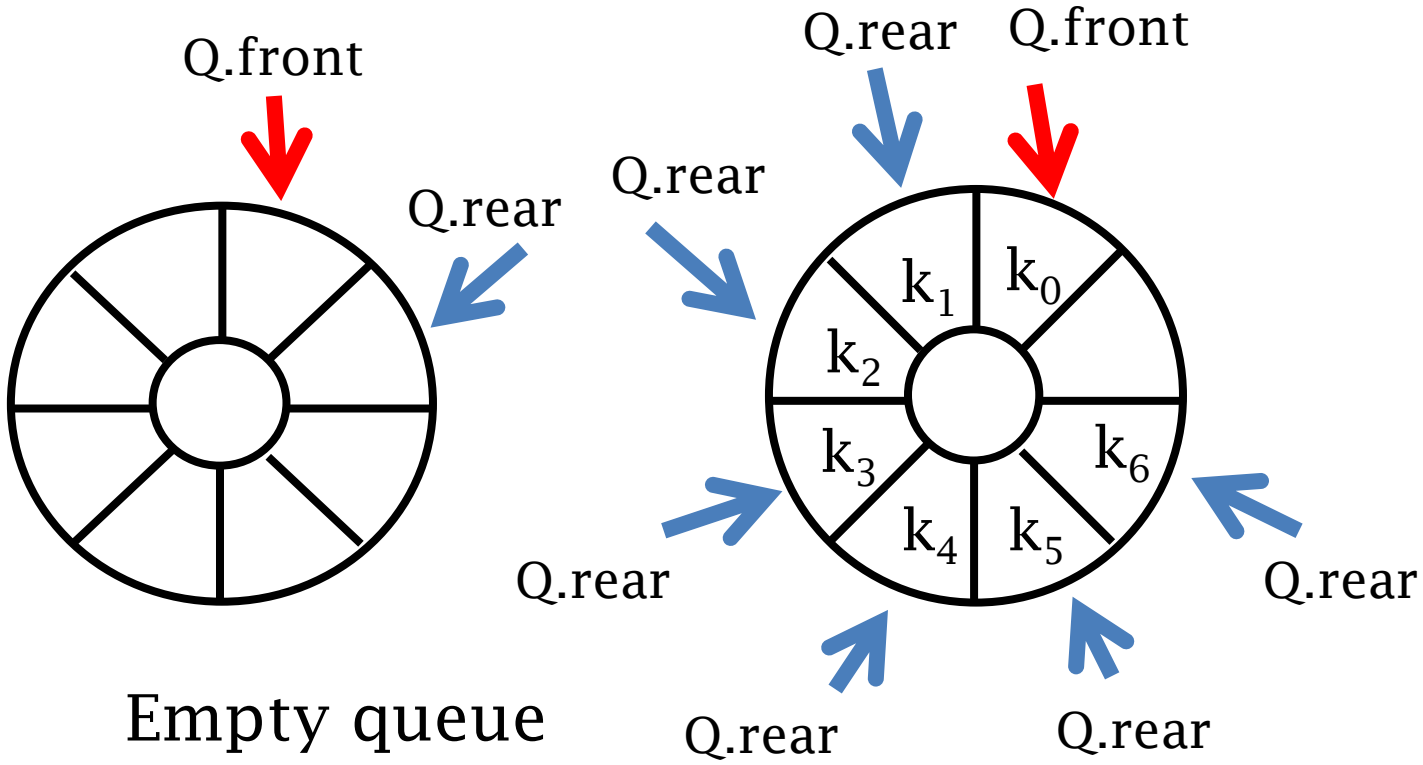
```
template <class T> class Queue {
public:                    // operation set of the queue
   void clear();           // change into empty queue
   bool enQueue(const T item); // insert item into the end of the queue, return true if
succeed, otherwise return false
   bool deQueue(T & item) ;
    // return the front element of the queue and remove it,return true if succeed
   bool getFront(T & item);
    // return the front element of the queue and do not remove it,return true if succeed

   bool isEmpty(); // return true if the queue is empty
   bool isFull();  // return true if the queue is full
};
```

Implementation of Queues

- **Sequential queue**
  - **The key point** is how to prevent false overflow
- **Linked queue**
  - Use single linked list to store, every element in the queue corresponds to a node in the linked list

Queue : Ring(true pointers)

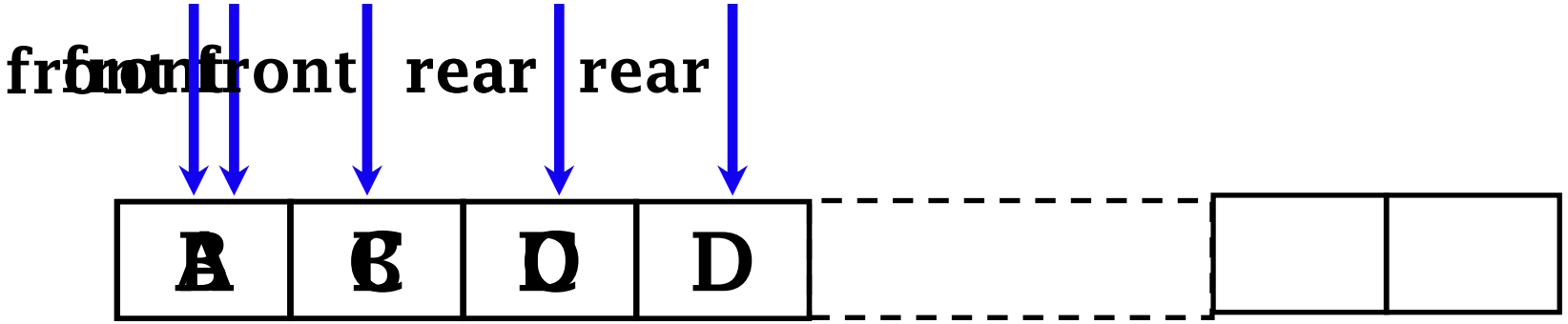


Class definition of sequential queues

```
class arrQueue: public Queue<T> {
private:
    int mSize;              // The size of array to store the queue
    int front;              // Subscript used to show the position of the front of the
queue
    int rear;               // Subscript used to show the position of the end of the
queue
    T * qu;                 // Array used to put queue elements of type T
public:                     // operation set of the queue
    arrQueue(int size);     // create an instance of the queue
    ~arrQueue();            // delete the instance and release space
}
```

The maintenance of sequential queue

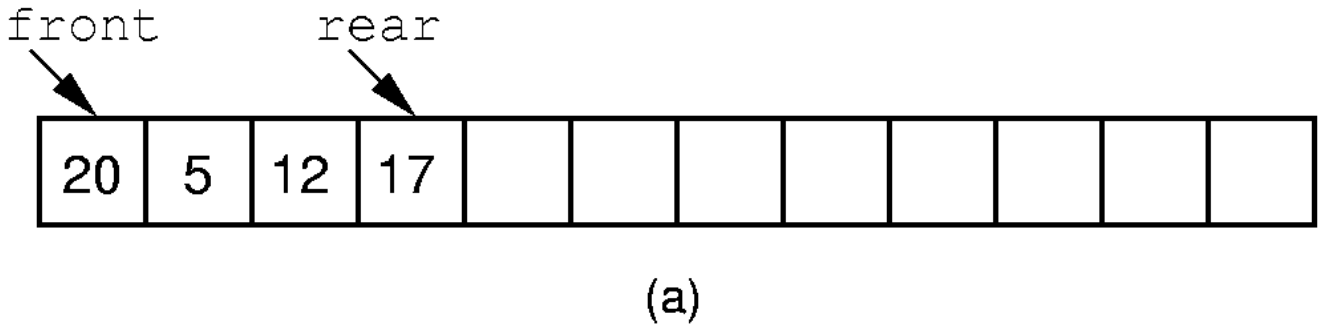
- Rear refers to



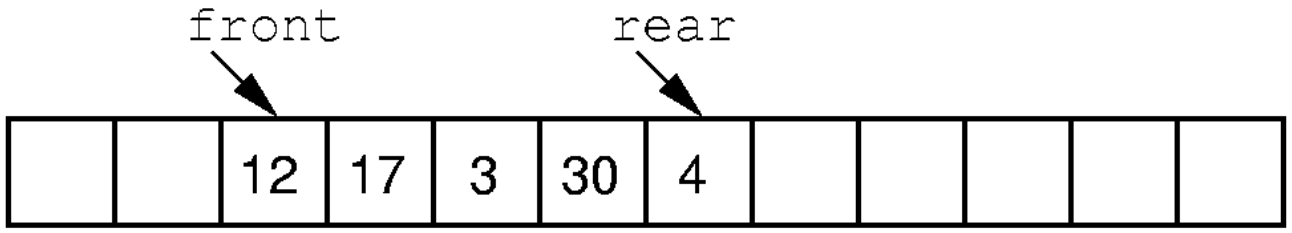
insert delete

The maintenance of sequential queue

- Front and rear are all true pointers



(a)



(b)

Implementation code of sequential queues

```
template <class Elem> class Aqueue : public Queue<Elem> {
private:
   int size;                  // The maximum capacity of queue
   int front;                 // The pointer of the front element of the queue
   int rear;                  // The pointer of the end element of the queue
   Elem *listArray;           // The array that store the elements
public:
   AQueue(int sz=DefaultListSize) {
      // Let the array for storage leave one more empty place
      size = sz+1;// size is the length of the array , and the max length of queue sz
      rear = 0; front = 1;     // you may assign rear=-1; front=0
      listArray = new Elem[size];
   }
   ~AQueue() { delete [] listArray; }
   void clear() { front = rear+1; }
```

Implementation code of sequential queues

```
bool enqueue(const Elem& it) {
    if (((rear+2) % size) == front) return false;
                        // There is only one empty place for the queue to be full
    rear = (rear+1) % size; // It needs to be moved to the next empty place first
    listArray[rear] = it;
    return true;
}
bool dequeue(Elem& it) {
    if (length() == 0) return false;
                              // the queue is empty
    it = listArray[front];      // move out of the queue first and then move the front subscript
    front = (front+1) % size;   // Increase in the formula of ring
    return true;
}
```

Implementation code of sequential queues

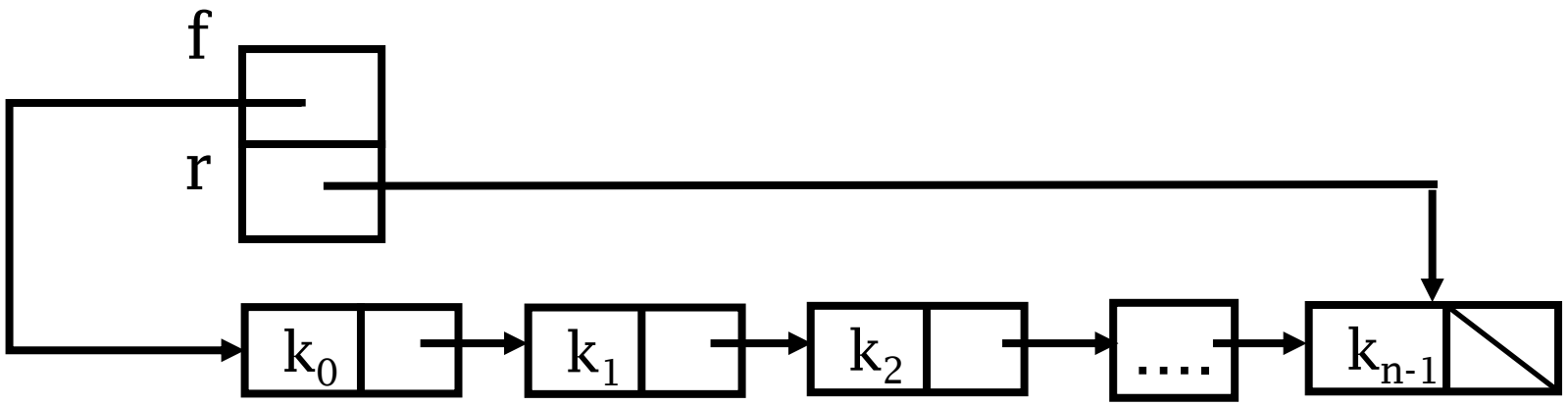
```
  bool frontValue(Elem& it) const {
   if (length() == 0)
            return false;      // the queue is empty
   it = listArray[front]; return true;
}
int length() const {
   return (size +(rear - front + 1)) % size;
}
```

Questions

- 1. You are given a queue with the length of n, you can just use the variable of front and rear, what is the largest number of elements that the queue can contain? Please give details of the derivation.
- 2. If you don't want to waste storage unit of the queue, what kind of other methods can you use ?

Representation of linked queues

- Singed linked list queue
- The direction of the linked pointer if from the front of the queue to the end of the queue



Class definition of linked queues

```
template <class T>
class lnkQueue: public Queue<T> {
private:
    int size;                       // the number of elements in the queue
    Link<T>* front;                 // the pointer of the front element of the queue
    Link<T>* rear;                  // the pointer of the end element of the queue
public:                             // operation set of the queue
    lnkQueue(int size);             // create an instance of the queue
    ~lnkQueue();                    // delete the instance and release space
}
```

Implementation code of linked queues

```
bool enQueue(const T item) {
//  insert the element to the end of the queue
   if (rear == NULL)       {      //if the queue is empty
          front = rear = new Link<T> (item, NULL);
   }
   else  {                  // add new elements
          rear-> next = new Link<T> (item, NULL);
          rear = rear ->next;
   }
   size++;
   return true;
}
```

Implementation code of linked queues

```
bool deQueue(T* item)  {
// return the front element of the queue and remove it
    Link<T> *tmp;
    if (size == 0)  {
// the queue is empty and no elements can be bring out of the
queue
        cout << “The queue is empty" << endl;
        return false;
    }
    *item = front->data;
    tmp = front;
    front = front -> next;
    delete tmp;
    if (front == NULL)
        rear = NULL;
    size--;
    return true;
}
```

Comparison between sequential queue and linked queue

- **Sequential queue**
  - Fixed storage space
- **Linked queue**
  - Use in the cases when the maximum size cannot be estimated

Both of them are not allowed to access internal elements of the queue

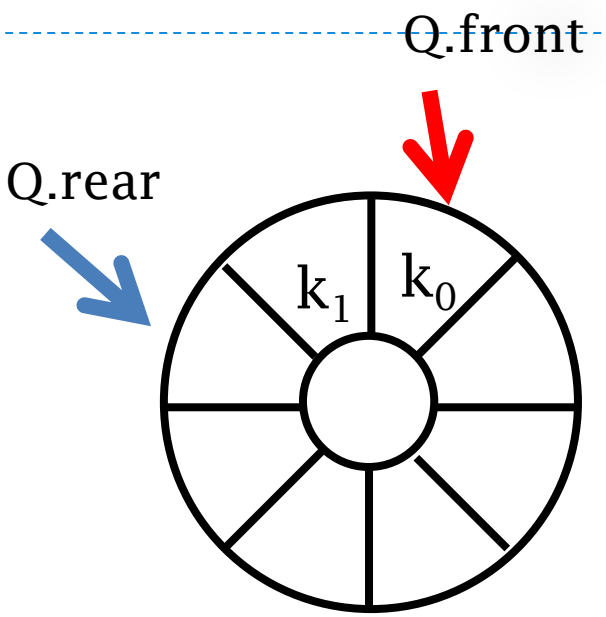
Applications for queues

- All the applications that meet the characteristics of FIFO can use queue as the way of data organization or intermediate data structure
- Scheduling or buffering
  - Message buffer
  - Mail buffer
  - The communication between computer hardware equipment also need queue as a data buffer
  - Resource management of operating system
- BFS

Questions

- Linked list are usually implemented by using linked list, why not use doubly linked list?

- And, if we apply false-pointers to a tail of a sequential queue, what is the difference from the case of true-pointers we have introduced?

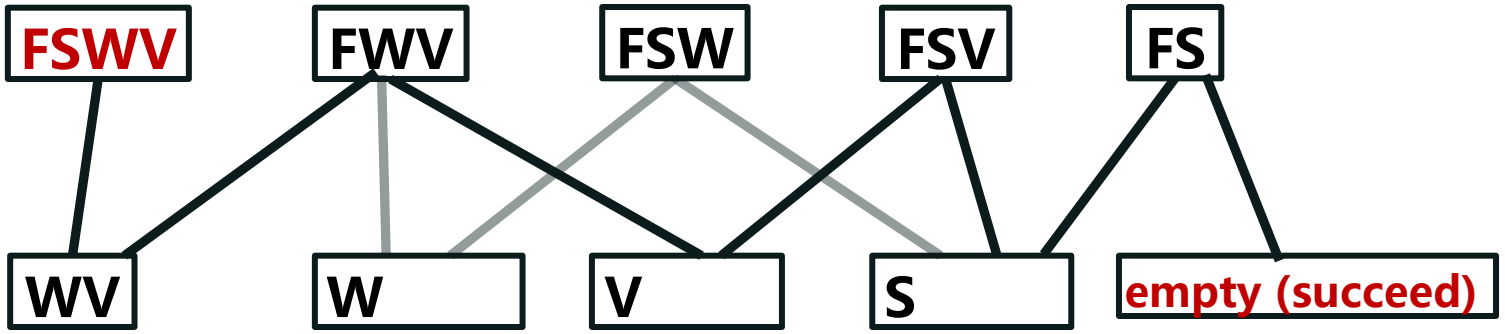


Chapter 3 Stacks and Queues

- Stacks
- Applications of Stacks
- Queues
  - Applications of Queues

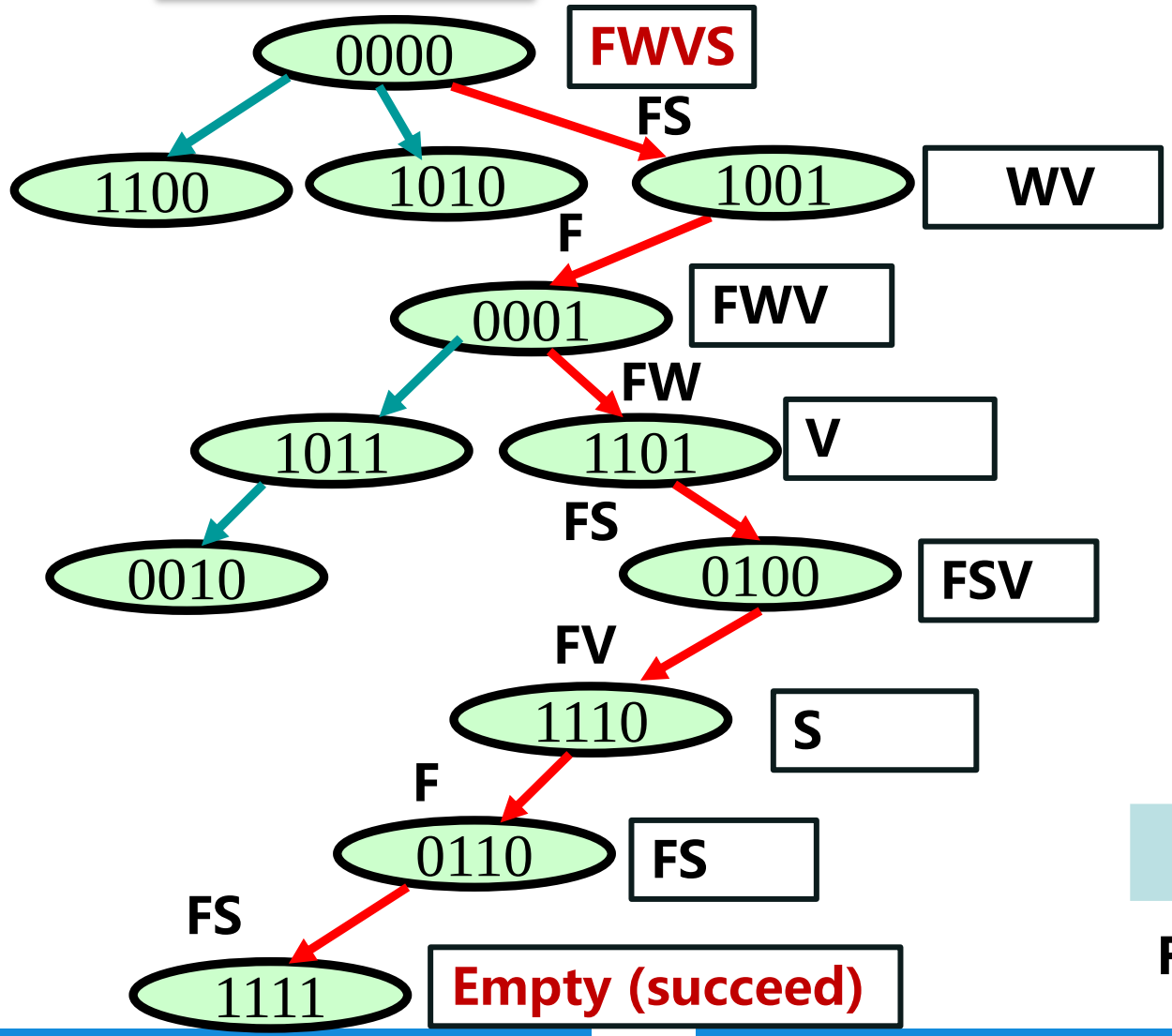
Farmer across the river

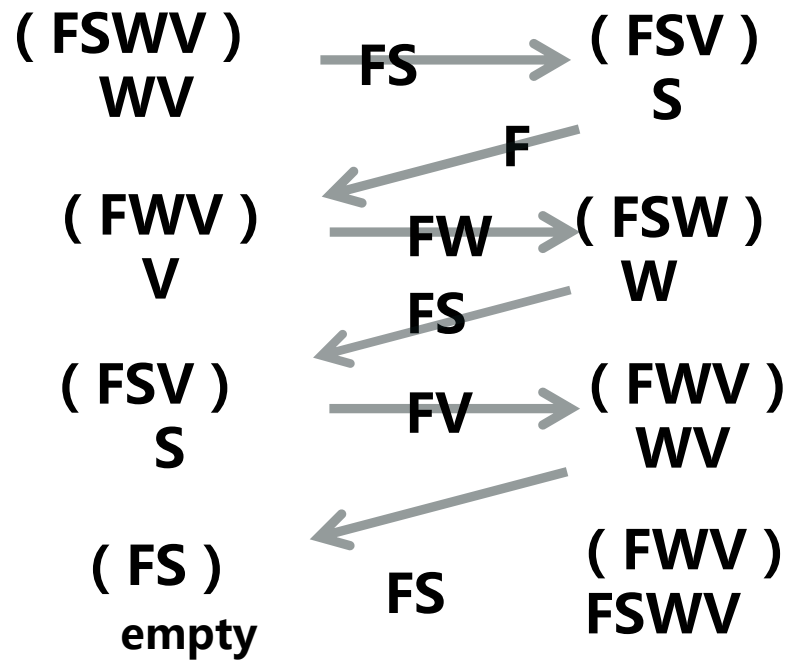
- **Problem abstraction**：FSWV boat across the river
  - Only the farmer can row the boat
  - There are only two positions on the boat include the farmer
  - Wolf and sheep, sheep and vegetables can not stay along without the farmer beside



Farmer is abbreviated as F
Sheep is abbreviated as S
Wolf is abbreviated as W
Vegetable is abbreviated as V

Application of queues

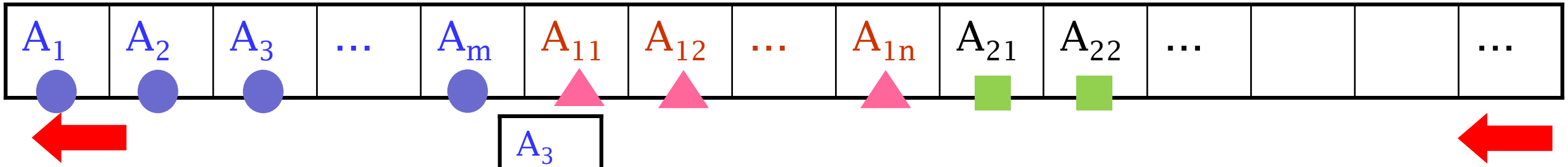




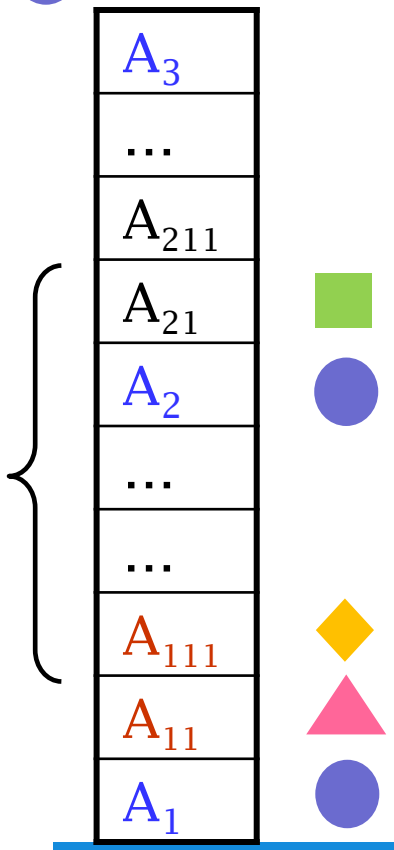
0	1	0	1
Farmer	Wolf	Vegetable	Sheep

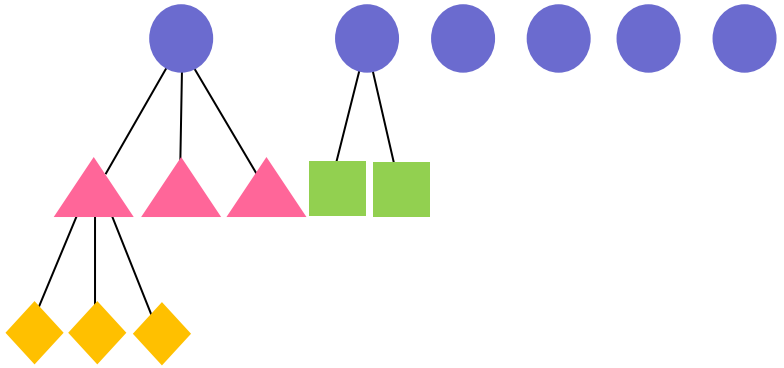
Problem Analysis

BFS : (m states)



DFS : (m states)





Data abstraction

- The state of each role is represented by their positions
  - Farmer、wolf、vegetable and sheep , each position is represented by a bit (Their positions are supposed to be in the order of FWVS) . If the target is in the original bank, the bit will be zero. And the bit will be one if the target is in the opposite bank

0	1	0	1

  - E.g. 0101 represents that farmer and vegetable are in the original bank , while wolf and sheep are in the opposite bank (This state is unsafe)

Representation of data

- Use status (integer) to represent the above four bit binary description of the state
  - The state 0x08 represents

1	0	0	0

  - The state 0x0F represents

1	1	1	1

- How to get the position of each role from the status above?
  - If function returns with value 1 , it means the person or thing you observe is in the original bank
  - Or it means the person or thing you observe is in the opposite bank

The function that determines the position of each role

```
bool farmer(int status)
{  return ((status & 0x08)  != 0);     }

bool wolf(int status)
{  return ((status & 0x04)  != 0);     }

bool cabbage(int status)
{  return ((status & 0x02)  != 0);     }

bool goat(int status)
{  return ((status & 0x01)  != 0);     }
```

F	W	V	S
1	x	x	x
x	1	x	x
x	x	1	x
x	x	x	1

F	W	V	S
0	1	0	1

Judge of safe state

```
//return true if safe , return false  if unsafe
bool safe(int status) {
    if ((goat(status) == cabbage(status)) &&
        (goat(status) != farmer(status)))
            return(false);          // sheep eat vegetables
    if ((goat(status) == wolf(status)) &&
        (goat(status) != farmer(status)))
            return(false);          // wolf eats sheep
    return(true);                   // The state left are safe
}
```

Algorithm abstraction

- The problem changed into: from state 0000 (integer 0) start , find state sequence made up of all the safe states , and takes the state 1111 as the final target.
  - Every state in the state sequence can be reached from its prior state by the action of farmer rowing across the river(one thing can be taken with him)
  - Repeated state can not appear in the sequence

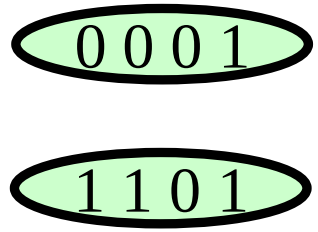
Algorithm design

- Define an integer queue moveTo , each element of it represents an intermediate state that can be reached safely
- You need to design another structure to record all the state that has been visited , and the path that has been find to be able to reach the current state
  - Use the ith element of the sequential table route to record whether state I has been visited
  - If route[i] has been visited , then record a precursor state value. And it represents unvisited if its value is -1
  - The length of route is 16

Implementation of the algorithm

```
void solve() {
    int movers, i, location, newlocation;
    vector<int> route(END+1, -1);
     // record the state path that has been considered
    queue<int> moveTo;
    // prepare the initial value
    moveTo.push(0x00);
    route[0]=0;
```

Implementation of the algorithm

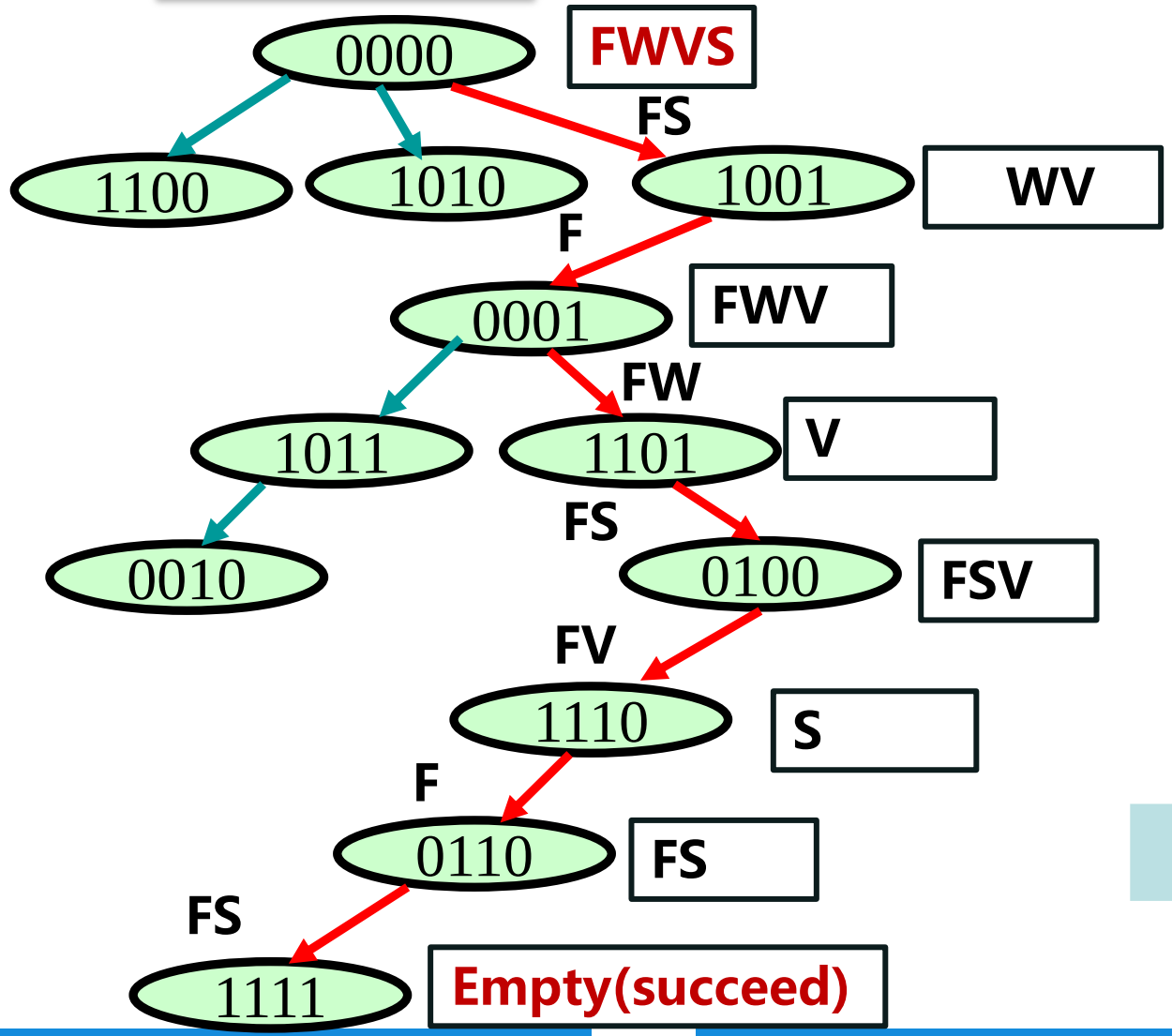


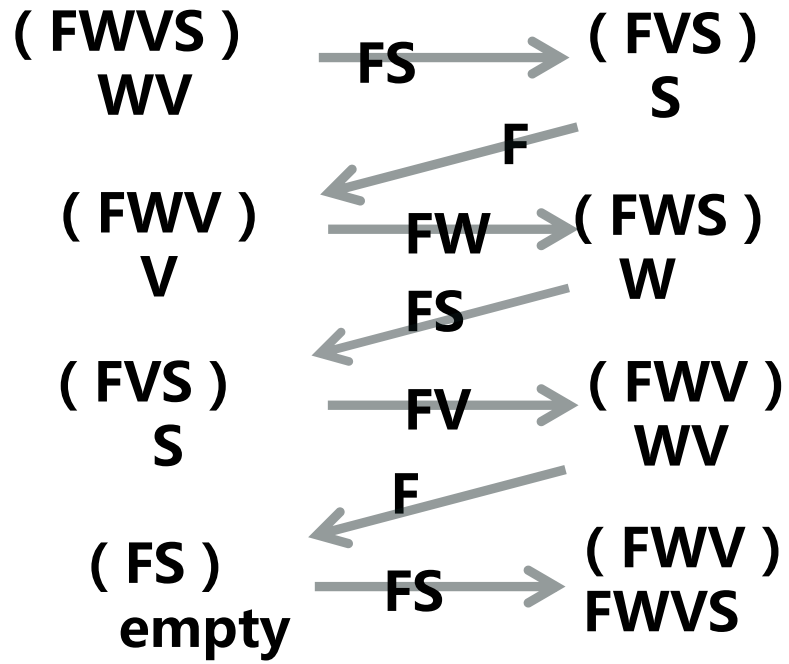

```
while (!moveTo.empty() && route[15] == -1) {
    // get the current state
    status = moveTo.front();
    moveTo.pop();
    for (movers = 1; movers <= 8; movers <<= 1) {
    // the farmer is moving all the time,
    // and only things in the same side of bank can move with him
        if (farmer(status)  == (bool)(status & movers)) {
            newstatus = status ^ (0x08 | movers);
            // Ways that are safe and not considered before
            if (safe(newstatus) && (route[newstatus] == -1)) {
                route[newstatus] = status;
                moveTo.push(newstatus); }
        }
    }
}
```

Implementation of the algorithm

```
    // print the path in the opposite direction
    if (route[15] != -1) {
        cout << "The reverse path is : “ << endl;
        for (int status = 15; status >= 0; status = route[status]) {
            cout << "The status is : “ << status << endl;
            if (status == 0) break;
        }
    }
    else
        cout << "No solution.“ << endl;
}
```

Applications of Queues

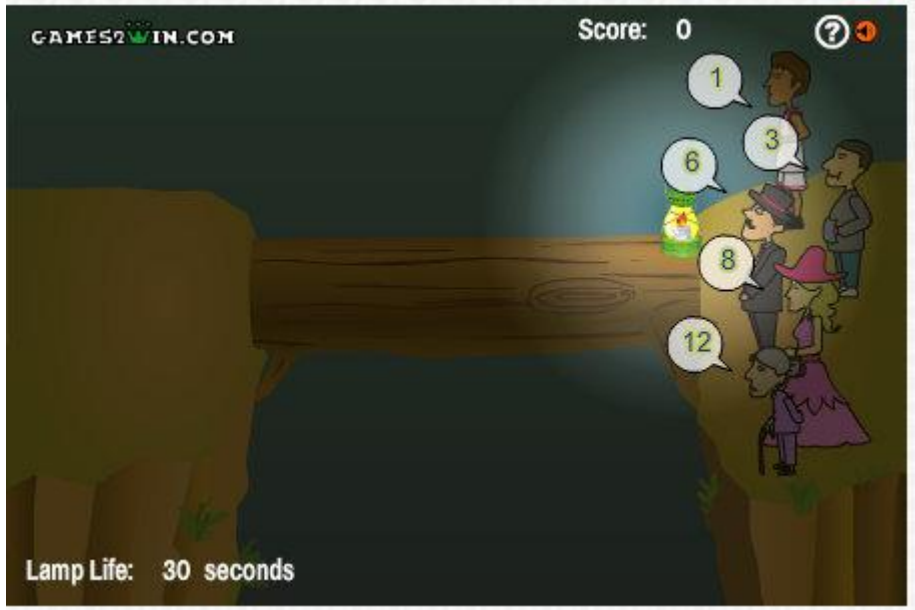




0	1	0	1
F	W	V	S

Question : Another small game

- Five people across the bridge with a lamp :
  - There is a lamp that can be used for just 30seconds , they have to go across the bridge before the lamp goes out
  - The speed of these five people is different : it takes the older brother 1 second , the younger brother 3seconds , the father 6 seconds , the brother 8 seconds and the grandmother 12 seconds to across the bridge
  - Only two people can go across the bridge at one time. And when they go across the bridge, one of them must send the lamp back to the original bank.



Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

Ming Zhang, Tengjiao Wang and Haiyan Zhao
Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)