به نام خدا

ساختمان داده ها

جلسه هفتم

دانشگاه صنعتی همدان

گروه مهندسی کامپیوتر

نیم سال دوم 98-1397

فصل سوم

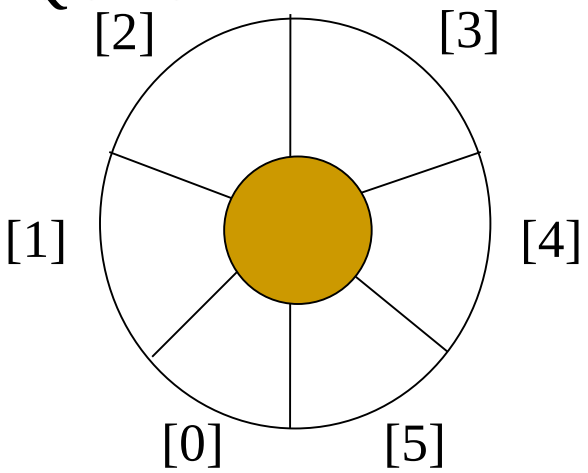
پشته و صف

Stack & Queue

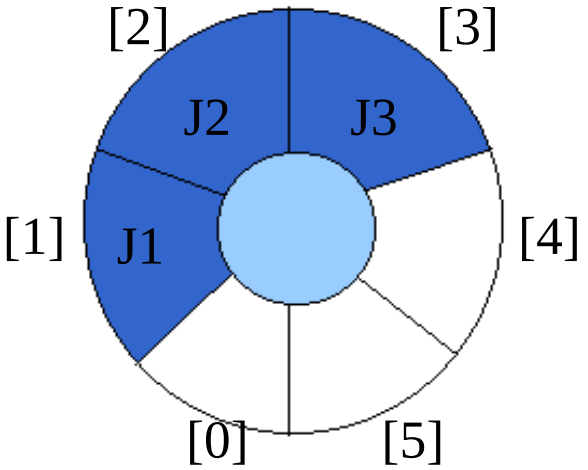
معایب روش آرایه ای

- در این روش اعضا به ترتیب وارد صف می شوند و به ترتیب از انتهای صف حذف می شوند ولی فضای عناصری که حذف می شوند دوباره استفاده نمی شوند و صف بعد از مدتی دیگر فضای خالی نخواهد داشت:
- دو روش برای حل این معضل وجود دارد:
  - شیفت دادن عناصر
  - استفده از صف دایره ای

EMPTY QUEUE

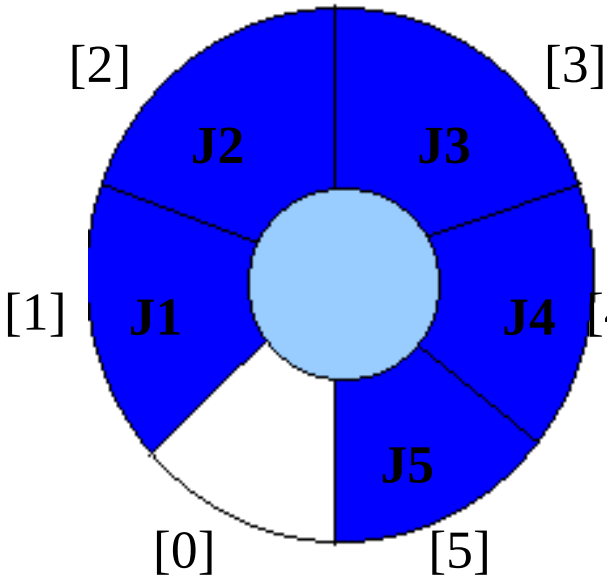


front = 0
rear = 0



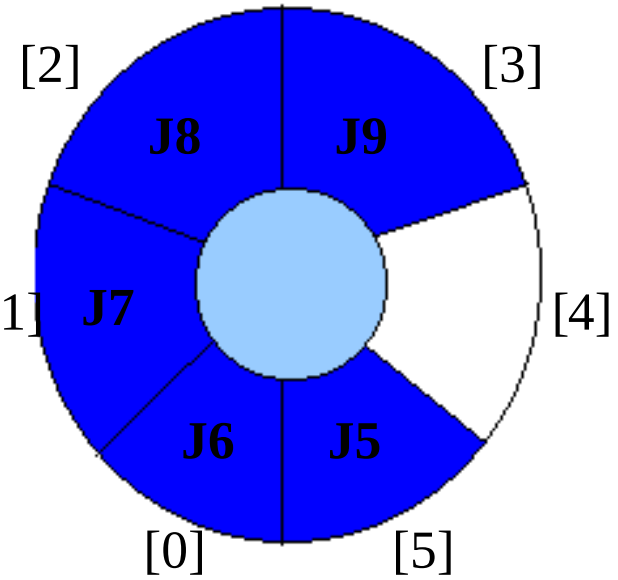
front = 0
rear = 3

FULL QUEUE



front =0
rear = 5

FULL QUEUE



front =4
rear =3

پیاده سازی صف حلقوی

```
private:
    int front;
    int rear;
    KeyType *queue;
    int MaxSize;
```

سازنده کلاس

```
template <class KeyType>
Queue<KeyType>::Queue (int MaxQueueSize) :
MaxSize(MaxQueueSize)
{
    queue = new KeyType[MaxSize];
    front = rear = 1;
}
```

توابع پر بودن و خالی بودن صف

```
template <class KeyType>
Boolean  Queue<KeyType>::IsFull()
{
    if (rear == MaxSize -1) return TRUE;
    else return FALSE;
}

template <class KeyType>
Boolean  Queue<KeyType>::IsEmpty()
{
    if (front == rear) return TRUE;
    else return FALSE;
}
```

حذف و اضافه کردن در صف حلقوی

```
template <class KeyType>
void Queue<KeyType>::Add (const KeyType& x)
// add x to the queue
{
    int k = (rear + 1) % MaxSize;
    if (front == k) QueueFull();
    else queue[rear = k] = x;
}

template <class KeyType>
KeyType* Queue<KeyType>::Delete (KeyType& x)
// remove and return top element from queue
{
  if (front == rear) {QueueEmpty(); return 0;}
    x = queue[++front %= MaxSize];
    return &x;
}
```

```
void main()
{
    Queue<int> s(5);
    int x;
    s.Add(5);
    s.Add(7);
    s.Delete(x);
    s.Add(9);
    s.Add(10);
    s.Delete(x);
    s.Delete(x);
    s.Delete(x);
}
```

مثالهایی از کاربردهای پشته و صف

ارزیابی عبارتهای ریاضی

- نحوه نمایش عبارتهای ریاضی
  - روش میانوندی nfix A+B
  - روش پسوندی Postfix AB+
  - روش پیشوندی Prefix +AB

مزیت روشهای پیشوندی و پسوندی این است که پرانتزها و کروشه ها دیده نمی شوند و اولویت اعمال هم وجود ندارد و اعمال به ترتیب قرار گرفتن اجرا می شوند.

بنابر این کامپایلر ها برای ارزیابی عبارتهای ریاضی انها را به یکی از فرمهای پسوندی یا پیشوندی تبدیل می کنند.

تبدیل عبارتهای میانوندی به پسوندی

Example: A / B – C + D * E – A * C

```
- ( ( ( ( A / B ) - C ) + ( D * E ) ) - ( A * C ) )
- ( ( ( ( A   B /   C -   ( D   E * +     A   C * -
- A B / C - D E * + A C * -
```

برای تبدیل دستی یک عبارت میانوندی به پسوندی به صورت زیر عمل می کنیم:

۱- عبارت را به صورت کامل پرانتز گذاری می کنیم

۲-عملگرها را به سمت راست حرکت می دهیم و جایگزین پرانتزهای سمت راست می کنیم

۳- تمام پرانتزها را حذف می کنیم

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

تبدیل عبارتهای میانوندی به پسوندی

- آیا این روش برای یک الگوریتم کامپیوتری هم موثر است؟

می توان روش بهتری ارایه کرد :

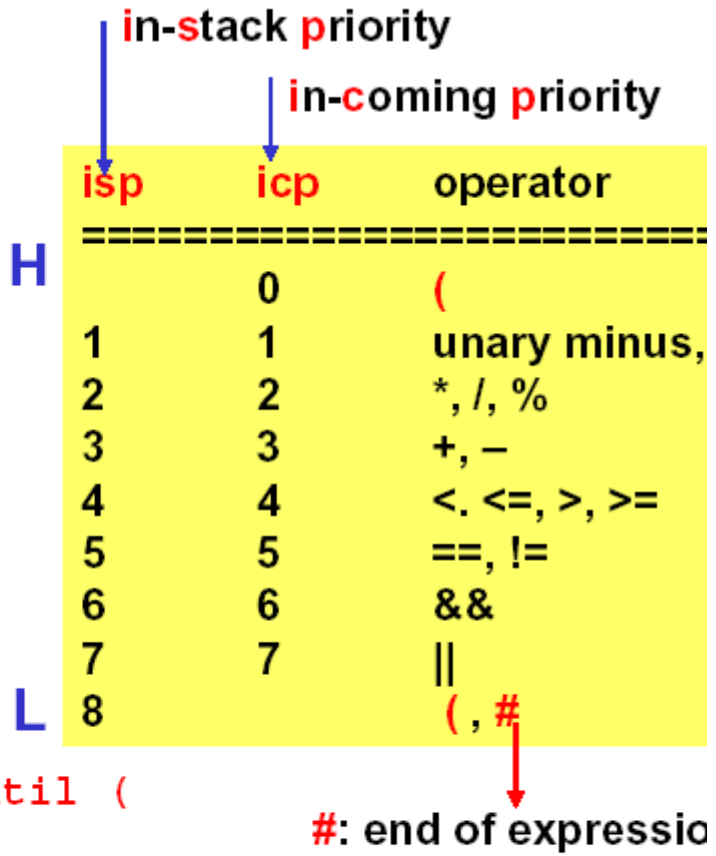
- عبارت پسوندی را از چپ به راست جمله به جمله (توکن به توکن) در نظر می گیریم
- اگر توکن مورد نظرعملوند بود انرا در خروجی می نویسیم
- اگر توکن مورد نظر عملگر بود
  - اگر اولویت ان از اولویت عنصر بالای پشته بیشتر بود در پشته قرار می دهیم
  - اگر اولویت آن کمتر یا مساوی عنصر بالای پشته بود عنصر بالای پشته را خارج کرده و در خروجی می نویسیم. این عمل انقدر تکرار می شود تا اولویت توکن از اولویت عنصر بالای پشته بیشتر شود.

- (در مورد اولویتها یک استثنا در مورد پرانتز وجود دارد: پرانتزچپ داخل پشته کمترین اولویت دارد و پرانتز راست باعث بیرون آمدن آن می شود و پرانتز چپ بیرون پشته بالاترین اولویت را دارد)

مثال

Example: A*(B+C)/D

next token	stack	output
none	#	none
A	#	A
*	#*	A
(	#*(	A
B	#*(	AB
+	#*(+	AB
C	#*(+	ABC
)	#*	ABC+ // pop until (
/	#/	ABC+*
D	#/	ABC+*D
done		ABC+*D/# // empty the stack



تابع تبدیل عبارت میانوندی به پسوندی

```
void postfix (expression e)
// Output the postfix form of the infix expression e. NextToken
// and stack are as in function eval (Program 3.18). It is assumed that
// the last token in e is '#.' Also, '#' is used at the bottom of the stack
{
   Stack<token> stack; // initialize stack
   token y ;
   stack.Add('#');
   for (token x = NextToken(e) ; x != '#' ; x = NextToken(e))
   {
      if (x is an operand)  cout << x ;
      else if (x == ')')  // unstack until '('
         for (y = *stack.Delete (y); y != '('; y = *stack.Delete (y)) cout << y ;
      else { // x is an operator
         for (y = *stack.Delete (y); isp (y) <= icp (x); y = *stack.Delete (y)) cout << y ;
         stack.Add(y);   // restack the last y that was unstacked
         stack.Add(x);
      }
   }

   // end of expression; empty stack
   while (!stack.IsEmpty()) cout << *stack.Delete(y) ;
} // end of postfix
```

ارزیابی یک عبارت پسوندی

```
void eval(expression e)
// Evaluate the postfix expression e. It is assumed that the last token (a token
// is either an operator, operand, or '#') in e is '#.' A function NextToken is
// used to get the next token from e. The function uses the stack stack
{
   Stack<token> stack ; //initialize stack
   for (token x = NextToken (e) ; x != '#'; x = NextToken (e))
      if (x is an operand) stack.Add(x) // add to stack
      else { // operator
         remove the correct number of operands for operator x from stack; perform the
         operation x and store the result (if any) onto the stack;
      }
```

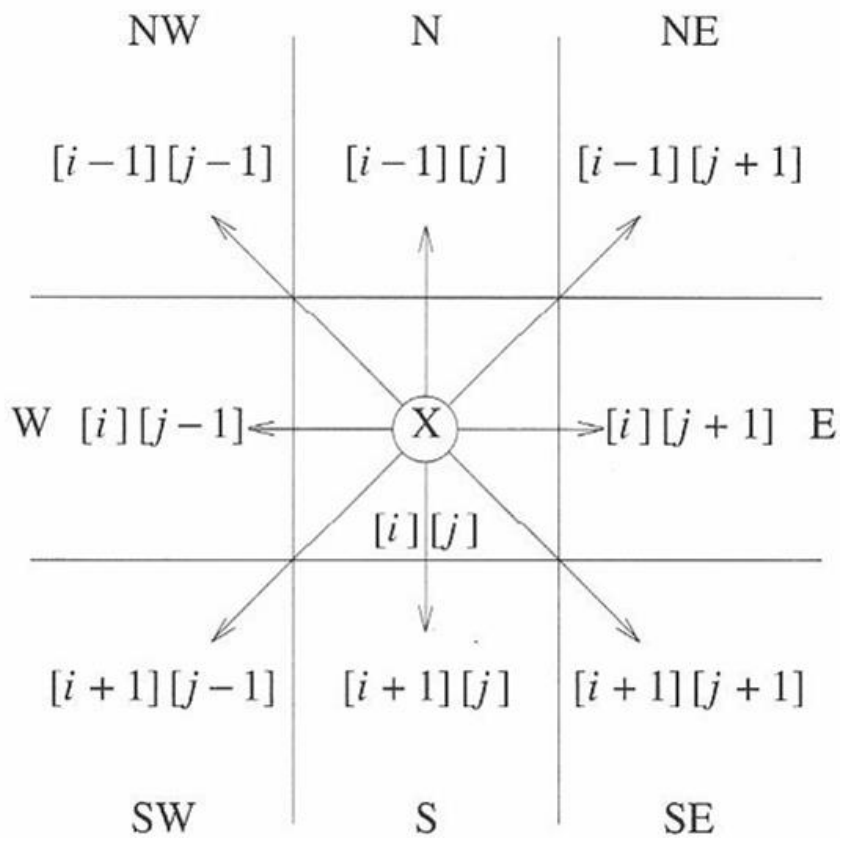
بازی MAZE

نقشه بازی Maze در ماتریس maze[m][p] ذخیره شده است و هر یک نشان دهنده دیوار است.

entrance

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

exit



q	*move* [*q*].*a*	*move* [*q*].*b*
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

- برای پیاده سازی حرکت به هشت جهت از این ساختار استفاده می کنیم:

```
Struct offsets {int a, b;}
enum{N, NE, E, SE, S, SW, W, NW}
offsets  mov[8];
struct items {  int x, y, dir;};
```

مثلا برای حرکت از نقطه i,j به اندازه یک گام به سمت جنوب غربی می توانیم داشته باشیم:

```
g=i+mov[sw].a;    h=j+mov[sw].b;
```

چگونه یک مسیر برای Maze پیدا کنیم؟

- در هر نقطه تمام جهت های ممکن از شمال در جهت عقربه های ساعت را بررسی می کنیم.

- وقتی یک جهت حرکت که به دیوار برخورد نمی کند پیدا شد بریا اینکه مسیر برگشت را از دست ندهیم مکان جاری و جهت حرکت جدید را در پشته ذخیره می کنیم

- برای انکه دوباره این نقطه را طی نکنیم ماتریس دیگری با نام mark[m][p] برای ثبت نقاط طی شده در نظر می گیریم و خانه متناظر ان نقطه در این ماتریس را یک می کنیم.

- هنگامی که در یک نقطه هیچ مسیری پیدا نشد به یک قدم عقب تر بر می گردیم (با مراجعه به پشته)

الگوریتم پیدا کردن مسیر

```
initialize stack to the maze entrance coordinates and direction east;
while (stack is not empty)
{
 (i,j,dir) = coordinates and direction deleted from top of stack ;
 while (there are more moves)
 {
  (g,h) = coordinates of next move ;
  if (( g == m ) && ( h == p )) success ;
  if ( (!maze [g ][h]) // legal move
            && (!mark [g ][h]) // haven't been here before
  {
        mark [g ][h] = 1 ;
        dir = next direction to try ;
        add (i,j,dir) to top of stack;
        i = g; j = h; dir = north;
  }
 }
}
cout << "no path found" << endl ;
```

```
void path(int m, int p)
{  mark[1][1] = 1;
  Stack<items> stack(m*p);  items temp;  temp.x = 1; temp.y = 1; temp.dir = E;
  stack.Add(temp);
  while (!stack.IsEmpty())
     {    temp = *stack.Delete(temp); // unstack
          int i = temp.x; int j = temp.y; int d = temp.dir;
          while (d < 8) // move forward
          {   outFile << i << " " << j << " " << d << endl;
              int g = i + move[d].a; int h = j + move[d].b;
              if ((g == m) && (h == p)) {
                 cout << stack;  cout << i << " " << j << endl;
                 cout << m << " " << p << endl;    return;
           }
       if ((!maze[g][h]) && (!mark[g][h])) {
              mark[g][h] = 1;   temp.x = i;  temp.y = j; temp.dir = d+1;
           stack.Add(temp);  i = g; j = h; d = N; }
       else d++; // try next direction
     }
  }
  cout << "no path in maze " << endl;}
```