Data Structures

Stacks

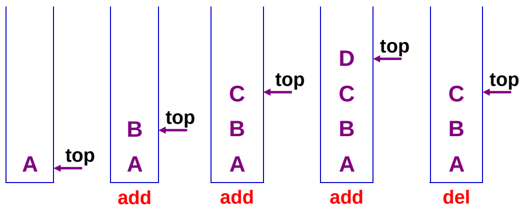
Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. http://inside.mines.edu/~dmehta/
4. ,

Stack

- **Linear list**
- **A LIFO (*Last-In-First-Out*) list**
- **One end is called top**
- **Other end is called bottom**
- **From the top only**
  - **Insertions / Additions / Puts / Pushes**
  - **Deletions / Removals / Pops**

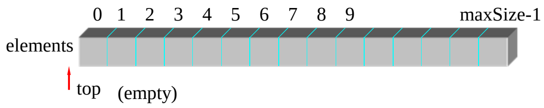
inserting and deleting elements in a stack:



Stack Presentment and Implement

- **Use**
  - **an array**
  - **a variable top**
    - **Initially, top = –1**

```
private:
    T* stack;
    int top;
    int capacity;  //maxSize
```



Stack elements are stored in stack[0] through stack[top]

The Class : Stack

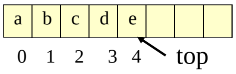
```
template<class T>
class Stack
{
   public:
    Stack(int stackCapacity = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const;
    T& Top() const;
    void Push(const T& item);
    void Pop();
   private:
    T *stack;      // array for stack elements
    int top;       // position of top element
    int capacity;  // capacity of stack array
};
```



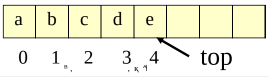
```
template <class T>
Stack<T>::Stack(int stackCapacity): capacity(stackCapacity)
{
   if (capacity < 1) throw "Stack capacity must be > 0";
   stack = new T[capacity];
   top = -1;
}

template <class T>
Inline bool Stack<T>::IsEmpty() const
{                                    // check whether top >= 0
   return(top == -1);
}
template <class T>
inline T& Stack<T>::Top()
{                                    // if not empty return stack[top]
   if (IsEmpty()) throw "Stack is Empty";
   return stack[top];
}
```

```
template <class T>
void Stack<T>::Push(const T& x)
{                                  // Add an element to the top of the stack
   if (top == capacity - 1)                  // if array full
   {
      ChangeSize1D(stack, capacity, 2*capacity);
      capacity *= 2;
   }
   stack[++top] = x;
}
```




```
template <class T>
void Stack<T>::Pop()
{                                  // Delete top element of stack
   if (IsEmpty()) throw "Stack is empty, cannot delete.";
   stack[top--];
}
```



Function **ChangeSize**

- use a 1D array to represent a stack
  - 1-Dimensional array
- changes the size from *oldSize* to *newSize*

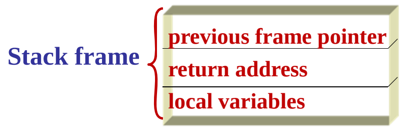
```
template <class T>
void ChangeSize(T* a, const int oldSize, const int newSize)
{
   if (newSize < 0) throw "New length must be >= 0";
   T* temp = new T[newSize];
   int number = min(oldSize, newSize);
   copy(a, a + number, temp);
   delete [] a;
   a = temp;
}
```


Application

- Recursion
- Try-Throw-Catch
- Parentheses Matching
- **Expressions**
- **Maze**
- Chess
- Switch Box Routing

System Stack and Recursion

- Be used by a program at runtime to process function calls
- A function is invoked
  - creates a structure : stack frame and activation-record
  - places it on the top of the system stack

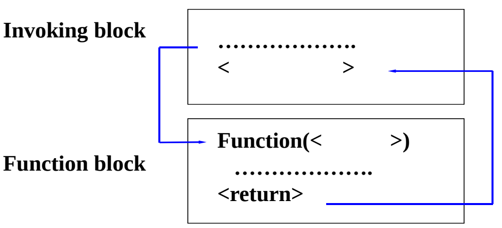


Activation-record

$$n! = \begin{cases} 1, & n = 0 \\ n*(n-1)!, & n \geq 1 \end{cases}$$

```
long Factorial(long n)
{
   if (n == 0) return 1;
   else return n*Factorial(n -1);
}
```


Activation-record



```
void main()
 {  int n;
   n = Factorial(4);
RetLoc1
}

long Factorial(long n)
{  int temp;
   if (n == 0) return 1;
   else temp = n * Factorial(n-1);
RetLoc2
   return temp;
}
```

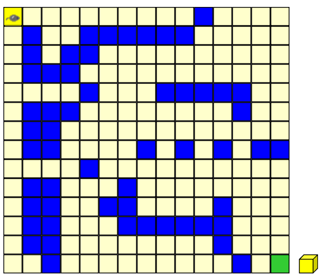

Data Structures

Application of Stacks : Mazing

Teacher : Wang Wei

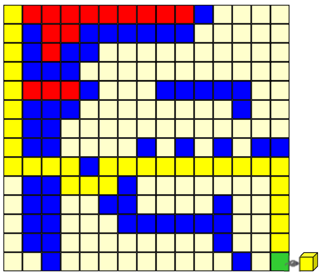
1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. http://inside.mines.edu/~dmehta/
4. ¬ ,

Rat In A Maze



- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



- **Path** from maze entry to current position operates as a stack.

 Standing… Wondering…

- Move forward whenever **possible**
  - no wall & not visited
- Move back ---- HOW ?
  - remember the footprints
  - or …… Better ?
  - NEXT possible move from previous position
- Storage ?
  - STACK

A Mazing Problem

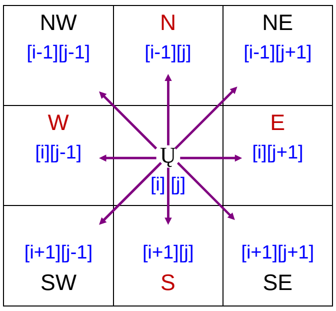
➢ **Find a path from the entrance to the exit of a maze**

entrance	**0**	1	0	0	1	1	0	1	1	
	1	**0**	**0**	1	0	0	1	1	1	
	0	1	1	**0**	1	1	1	0	1	
	1	1	0	**0**	1	**0**	**0**	1	0	
	1	0	0	1	**0**	1	1	**0**	1	
	0	0	1	1	0	1	**0**	1	1	
	0	1	0	0	1	1	**0**	**0**	**0**	**exit**

Representation

- maze[i][j] $1 \le i \le m$, $1 \le j \le p$
  - 1 --- blocked
  - 0 --- open
  - the entrance : maze[1][1]
  - the exit : maze[m][p]
  - current point : [i][j]
  - boarder of 1's,
    - so a maze[m+2][p+2]
  - 8 possible moves
    - N, NE, E, SE, S, SW, W, NW

To predefine the 8 moves




```
struct offsets
{ int a;
  int b;
};
```

```
enum directions {N, NE, E, SE, S, SW, W, NW};
offsets move[8];
```


The basic idea :

- ✓ Given *current* position [i][j] and 8 directions to go
- ✓ Pick one direction d
- ✓ Get the new position [g][h]
- ✓ If [g][h] is the goal, success
- ✓ If [g][h] is a legal position, save [i][j] and d+1 *in a stack*
  - ✓ in case, take *a false path* and need to try another direction
  - ✓ [g][h] becomes the new *current* position
- ✓ Repeat until either success or every possibility is tried

- **In order to prevent us from going down the same path twice :**
  - ✓ **mark[m+2][p+2] : use another array**
    - ✓ **which is initially 0**
  - ✓ **mark[i][j] : is set to 1 once the position is visited**
- **Need a stack of items:**

```
struct Items {
    int x, y, dir;
};
```

- **Set the size of stack to m*p**
  - to avoid doubling array capacity during stack pushing


```
void path(const int m, const int p)
{   //Output a path (if any) in the maze
    //maze[0][i]=maze[m+1][i]=maze[j][0]=maze[j][p+1]=1,0≤ i ≤ p+1, 0 ≤ j ≤ m+1
   // start at (1,1)
    mark[1][1]=1;
    Stack<Items> stack(m*p);
    Items temp(1, 1, E);
    stack.Push(temp);
    while ( !stack.IsEmpty() )
    {
         temp= stack.Top();
        Stack.Pop();
        int i=temp.x; int j=temp.y; int d=temp.dir;
```



```
        while (d<8)
        {
           int g=i+move[d].a; int h=j+move[d].b;
           if ((g==m) && (h==p)) {     // reached exit
               // output path
               cout <<stack;
               cout << i<<" "<< j<<" "<<d<< endl; // last two
               cout << m<<" "<< p<< endl;           // points
               return;
             }
```

```
            if ( (!maze[g][h]) && (!mark[g][h]) ) {  //new position
                mark[g][h]=1;
                temp.x=i;  temp.y=j;  temp.dir=d+1;
                stack.Push(temp);
                i=g ; j=h ; d=N;      // move to (g, h)
            }
            else d++;                 // try next direction
        }
    }
 cout << "No path in maze."<< endl;
}
```


Idea

- **scan expression from left to right**
- **when a left parenthesis is encountered, add its position to the stack**
- **when a right parenthesis is encountered, remove matching position from stack**

Data Structures

Application of Stacks : Expressions

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. http://inside.mines.edu/~dmehta/
4. ,

Arithmetic Expressions

How to generate machine-language instructions to evaluate an arithmetic expression ?

(a + b) * (c + d) + e – f/g*h + 3.25

- **Expressions comprise three kinds of entities**
  - **Operators : +, -, /, ***
  - **Operands : a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.**
  - **Delimiters : (,)**

Operator Degree

- **Number of operands that the operator requires**
- **Binary operator (ԋ˝Ḥ ƀ) requires two operands (2 ɷ ƀ)**
  - **Such as a + b , c / d , or e - f**
- **Unary operator (w̅˝Ḥ ƀ) requires one operand (1 ɷ ƀ)**
  - **Such as + g or - h**

Infix Form

- Normal way to write an expression
- Binary operators come in between their left and right operands
  - Such as

a * b

a + b * c

a * b / c

(a + b) * (c + d) + e – f/g*h + 3.25

Operator Priorities

- Such as

  priority(*) = priority(/) > priority(+) = priority(-)
- When an operand lies between two operators, the operand associates with the operator that has higher priority

优先级	操作符
1	负号(-)， ！
2	*， /， %
3	+， -
4	<， <=， >=， >
5	==， !=
6	&&
7	||

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left

  a + b - c

  a * b / c / d

- Sub-expression within delimiters is treated as a single operand, independent from the remainder of the expression
  - Such as **parentheses ( )**

    (a + b) * (c – d) / (e – f)

- **Postfix** and **Prefix** expression forms
  - it is **easier for a computer to evaluate expressions** that are in these forms
  - *do not rely on operator priorities, a tie breaker, or delimiters*

Postfix Form

- The postfix form of a variable or constant is the same as its infix form
  - **a, b, 3.25**
- The relative order of operands is the same in infix and postfix forms
- Operators come immediately after the postfix form of their operands
  - Infix : **a + b**
  - Postfix : **ab+**

Unary Operators

- Replace with new symbols

+ a => **a @**
+ a + b => **a @ b +**
- a => **a ?**
- a-b => **a ? b -**

Problem:

how to evaluate an expression?

Infix to Postfix

Idea: note the order of the operands in both infix and postfix

infix: A / B – C + D * E – A * C

postfix: A B / C – D E * + A C * –

immediately passing any operands to the output

store the operators somewhere until the right time

A*(B+C)*D → ABC+*D*

A*(B+C)*D → ABC+*D*

Next token	stack	output
A	#	A
*	#*	A
(	#*(	A
B	#*(	AB
+	#*(+	AB
C	#*(+	ABC
)	#*	ABC+
*	#*	ABC+ *
D	#*	ABC+ *D
#	#	ABC+ *D*

- **isp : in-stack priority ( )**
  icp : in-coming priority (/)

Operator x	#	(	-	*, /, %	+, -	)
isp	8	8	1	2	3	

```
// output the postfix of the infix expression e. It is assumed
// that the last token in e is '#'. Also, '#' is used at the bottom
// of the stack.
//
void Postfix (Expression e)
{
   Stack<Token> stack;              //initialize stack
   stack.Push('#');
```



```
  for  (Token x=NextToken(e); x!='#'; x=NextToken(e))
    if (x is an operand) cout<<x;
    else if (x==')')
      {                                      // unstack until '('
        for  (; stackTop()!='('; stack.Pop())
            cout<<stack.Top();
        stack.Pop();                          // unstack '('
        }
    else {                                    // x is an operator
       for (; isp(stack.Top()) <= icp(x); stack.Pop())
          cout<<stack.Top();
       stack.Push(x);
     }
  // end of expression, empty the stack
  for (; !stack.IsEmpty()); cout<<stack.Top(), stack.Pop());
  cout << endl;
}
```


Data Structures

Queues

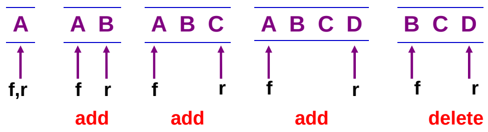
Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. http://inside.mines.edu/~dmehta/
4. ,

Queues

- Linear list
- A **FIFO** (*First-In-Fist-Out*) list
- One end is called **front**
- Other end is called **rear**
- Additions are done at the **rear** only
- Removals are made from the **front** only

The Queue

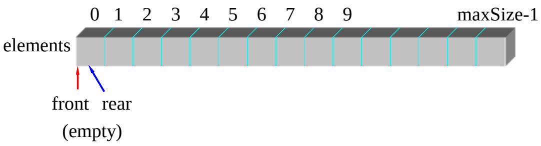


f = queue **front** **r** = queue **rear**

Queue Presentment and Implement

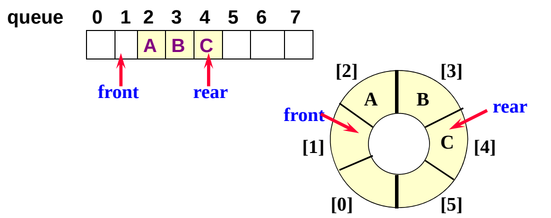
- **Use**
  - **an array**
  - **a circular representation**
  - **two variable *front* and *rear***
    - **Initially, *front = rear =* 0**

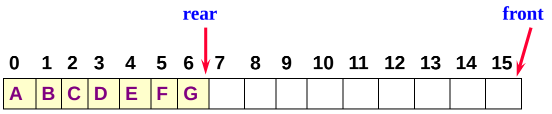
```
private:
    T* queue;
    int front,
    int rear,
    int capacity;   //maxSize
```



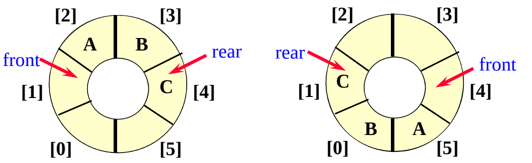
Queue elements are stored in queue[front] through queue[rear]

Custom Array Queue use a circular representation





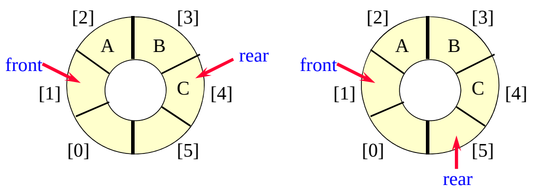
- Use integer variables front and rear
  - front is one position counter clockwise from first element
  - rear gives position of last element



- **Possible configuration with 3 elements**
- **Another possible configuration with 3 elements**

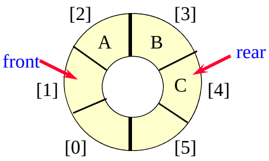
Push An Element

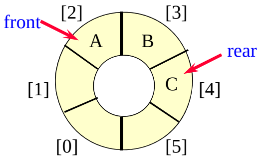
- Move rear one clockwise
- Then put into queue[rear]



Pop An Element

- Move front one clockwise

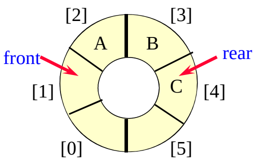




- Then extract from queue[front]

Moving rear Clockwise

```
rear++;
if (rear = = capacity) rear = 0;
```




```
rear = (rear + 1) % capacity;
```


Empty a Queue

- **When a series of removes causes the queue to become empty**
  - **front = rear**
- **When a queue is constructed, it is empty**
- **So initialize front = rear = 0**

Full a Queue

- **When a series of adds causes the queue to become full**
  - **front = rear**

- **So , cannot distinguish(≠) between a full queue and an empty queue**

```
template <class T>
inline T& Queue<T>::Front()
{
    if (IsEmpty()) throw "Queue is empty. No front element";
    return queue[(front+1)%capacity];
}

template <class T>
inline T& Queue<T>::Rear()
{
    if (IsEmpty()) throw "Queue is empty. No rear element";
    return queue[rear];
}
```



```
template <class T>
void Queue<T>::Pop()
{                           // Delete front element from queue
    if (IsEmpty()) throw "Queue is empty. Cannot delete";
    front = (front+1)%capacity;
    queue[front];
}
```

- **For the circular representation**
  - **the worst-case add and delete times are O(1)**
    - assuming no array resizing is needed


```
template <class T>
void Queue<T>::Push(const T& x)
{                                       // add x at rear of queue
    if ((rear+1)%capacity == front)
    {     // queue full,  double capacity
          // code to double queue capacity comes here
    }
    rear = (rear+1)%capacity;
    queue[rear] = x;
}
```