Stacks

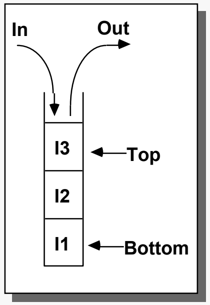
Slides

Stack Definition

Description

- Restricted list structure
- Dynamic LIFO Storage Structure
  - Size and Contents can change during execution of program
  - Last In First Out
- Single Type Element (not generic)

Only the top element may be accessed.



Insertion Order: I1, I2, I3

Operations

- **Stack** ( ) ;
  - set Stack to be empty
- bool **Empty** ( ) const;
  - check if stack is empty
- bool **Full** ( ) const;
  - check if stack is full
- bool **Push** (const ItemType& item) ;
  - insert item onto the stack
- Item **Pop** ( ) ;
  - remove & return the item at the top of the stack

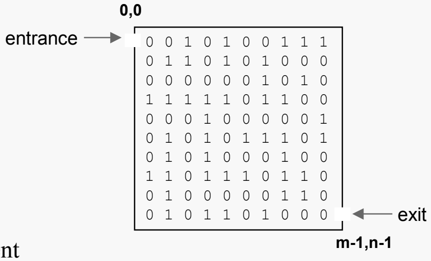
```
Stack S ;
S.Push(  I1 ) ;
S.Push ( I2 ) ;
S.Push ( I3 ) ;
```

Some implementations define:
Item Top( ) ;
Returns top item in the stack, but does not remove it.
Pop() ;
In this case removes the top item in the stack, but does not return it.

Maze Example

Problem

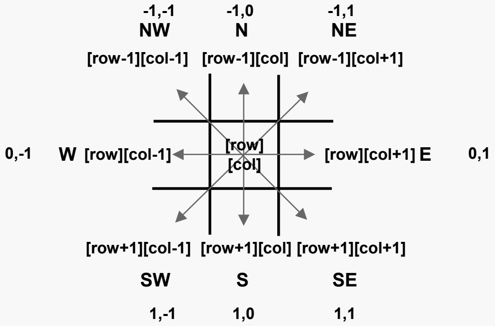
- Given a maze represented by a 2-Dim array, (where 0 = door, 1 = wall), with the entrance at the upper left & exit at the lower right find a path, if one exists, thru the maze.



```
0 0 1 0 1 0 0 1 1 1
0 1 1 0 1 0 1 0 0 0
0 0 0 0 0 0 1 0 1 0
1 1 1 1 1 0 1 1 0 0
0 0 0 1 0 0 0 0 0 1
0 1 0 1 0 1 1 1 0 1
0 1 0 1 0 0 0 1 0 0
1 1 0 1 1 1 0 1 1 0
0 1 0 0 0 0 0 1 1 0
0 1 0 1 1 0 1 0 0 0
```

Movement

- Any of the 8 possible compass points



Maze Example: Setup

Border

- Surround the maze by a border of walls (1's)
- Eliminates check for possible non-existant maze locations
- M x N maze requires (M+2) * (N+2) array locations
- Entrance is at [1][1]. Exit is at [M][N] (EXITROW, EXITCOL).

Initialize Moves

- Declare:

```
struct offsets  {  //                       direction
    int vert, hort;//  0     1      2     3     4     5      6      7
    };             //  N    NE     E     SE    S     SW     W      NW
offsets  move[8] = { -1,0, -1,1, 0,1, 1,1, 1,0, 1,-1, 0,-1, -1,-1};
```

- Movement Direction = 0 .. 7 corresponding to the directions setup in the move array.
- To determine the next location in direction dir:
  - `nextRow = row + move[dir].vert ;`
  - `nextCol = col + move[dir].hort ;`

Avoiding Getting Lost

- Declare a second 2-Dim array, `mark`, to store the maze paths already checked, (to avoid circular paths).
- Initialize all entries of mark to 0.
- Mark a position to 1 as it is visited.
- Moves into new squares are only allowed if their mark == 0.
  - Not previously visited

Stack Items

- stack Item: contains a row, col and direction
- represents the previous position and the next direction to take to move out of the previous position.
- Assume a stack item class, position, exists.

Maze Example: Path()

```
const int  MAXDIRECTIONS   =      8;
const int  DOOR            =      0;
const int  VISITED         =      1;

void Path ( const mazeType  maze ) {

  int currRow, currCol, nextRow, nextCol, dir;
  bool  found = false;
  Item position( 1, 1, 2 ); //initial position, dir-East
  mazeType mark  = {0};
  Stack    visitStk ;

  mark[1][1] = VISITED;
  visitStk.Push ( position );     //prime stack

  while ( (!visitStk.Empty()) && (!found) ) {
     position= visitStk.Pop ();
     currRow = position.GetRow();
     currCol = position.GetCol();
     dir = postion.GetDirection();//move in direction dir
     while ( (dir < MAXDIRECTIONS)  && (!found) ) {
        nextRow = currRow + move[dir].vert;
        nextCol = currCol + move[dir].hort;
        if ( (nextRow == EXITROW) &&
             (nextCol == EXITCOL) )
           found = true;
        else if ( (maze[nextRow][nextCol] == DOOR) &&
                ( !mark[nextRow][nextCol]) ) {
           mark[nextRow][nextCol] = VISITED;
           position.SetRow(currRow);
           position.SetCol(currCol);
           position.SetDir(++dir);//next dir if returning
           visitStk.Push ( position );//save old pos
           currRow  = nextRow;
           currCol = nextCol;
           dir = 0;  //start moving from new loc
        }
        else
           ++dir;
     } // end inner while
  } // end outer while
}// end Path
```

Maze Example: Output

Path Function (extension)

```
// Output Path Search Results

  if (found) { //output path thru maze in reverse steps

     cout << “The path is : ” << endl;
     cout << “row  col” << endl;
     cout << setw(2) << EXITROW
          << setw(5) << EXITCOL << endl;
     while ( ! visitStk.Empty() ) {
        position = visitStk.Pop();
        cout << setw(2) << position.GetRow()
             << setw(5) << position.GetCol() << endl;
     }
  else
     cout << “The maze does not contain a path out ”;

}  // end function
```

The visitStk holds the backtracking information that is used when no moves, (paths), exist out of the current position. The top item contains the previous position that is used to reset the current position when backtracking.

It must be able to hold a maximum of M x N positions, since each square containing a door is visited at most once and there can be no more than M x N doors in the maze.

Sequential Stack Implementation

Array Representation

- **Stack.h**

```
const int MAXSTACKSIZE= 100;
//typedef     arbitrary Item;
#include "Item.h"

class Stack {
private:
  int top;
  Item Items[MAXSTACKSIZE];
public:
  Stack();
  bool Empty( ) const;
  bool Full  ( ) const;
  bool Push  (const Item& Item);
  Item Pop( ) ;
};
```

Alternative Inefficient Implementation: top is fixed at the start of the array and the bottom *floats* thru the array.

Considerations

- Push and pop insert & remove elements from the array (stack) at the top location then increment & decrement top (count).
- top (count) contains the index of the array location 1 position ahead of where the actual top element is stored, (top == size).

Error Checks

- Stack Overflow: Attempt to Push on a full stack
- Stack Underflow: Attempt to Pop off of an empty stack

Stack Views

- Application: Maze program: history of moves
- User: Dynamic abstract stack
- Implementation: Static (maximum) size

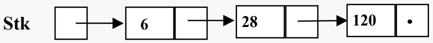
Another Implementation: Array[0] holds top index, only possible if the stack elements are integers.

Linked Stack Implementation

Linked-List Representation

- top is fixed at the head of the list
- Push & Pop operate only on the head of the list

Pointer Implementation



- top of stack == head list pointer

List Class Implementation

- Stack.h

```
#include "LinkList.h"
//typedef     arbitrary Itemtype;
#include "Item.h"

class Stack {
private:
  LinkList stk;

public:
  // Stack(); //LinkList constructor
  bool Empty( ) const;
  bool Full  ( ) const;
  bool Push  (const Item& Item);
  Item Pop( ) ;
};
```

Class aggregation

(implement using List Class operations)

Linked Stack Implementation

Stack.cpp

```
#include "Stack.h"

bool Stack::Empty( ) const {
  return ( stk.isEmpty() );
}

bool Stack::Full( ) const {
  Item*  newNode= new(nothrow) Item;
  if (newNode == NULL )
     return true;
  delete newNode;
  return false;
}

bool Stack::Push(const Item& Item) {
  return( stk.PrefixNode(Item) );
}

Item Stack::Pop( ) {
  Item temp;
  stk.gotoHead();
  temp = stk.getCurrentData();
  stk.DeleteCurrentNode();
  return( temp );
}
```

```
//if top() was to be implemented:

Item Stack::Top( ) {
  Item temp;
  stk.gotoHead();
  temp = stk.getCurrentData();
  return( temp );
}
```

Stack of Characters

String Representation

- Empty Stack == Empty String
- Top of Stack == End of String
- String operations are used to implement stack operations
  - Enforces stack behavior on strings of type stack
  - Maps one data structure, (stack), onto another, (string)

Headers

- Stack.h

```
#include <string>
typedef char Item;

class Stack {
private:
  string stk;
public:
  // Stack(); //string constructor
  bool Empty( ) const;
  bool Full  ( ) const;
  bool Push  (const Item& Item);
  Item Pop( ) ;
};
```

Interface unchanged

Stack of Characters: Operations

```
#include "Stack.h"
using namespace std;

bool Stack::Empty( ) const {
  return ( stk.empty() );
}

bool Stack::Full( ) const {
  return( stk.length() == stk.max_size() );
}

bool Stack::Push(const Item& Item) {
  stk = stk + Item;
  return ( Full() );
}

Item Stack::Pop( ) {
  Item temp;
  int i;

  i = stk.length();
  temp = stk.at(i-1);
  stk.erase(i-1, 1);
  return( temp );
}
```

```
//if top() was to be implemented:

Item Stack::Top( ) {
  Item temp;
  int i;

  i = stk.length();
  temp = stk.at(i-1);
  return( temp );
}
```

Iterative KnapSack

```
//Item.h

  enum Status  {NONE , INCLUDED , EXCLUDED };
  typedef Status Item;
```

```
//Iterative (nonrecursive) solution
#include "Stack.h"

bool Knap (const int ray[], int total, int start, int end)
{
   bool     found = false;
   Stack    StatusStack; //Stack of statuses
   Item    dummy;

   StatusStack.Push( NONE );

   do  {
     if ( (found) || (total == 0) ) { // soln found
       found = true ;
       --start;
       dummy = StatusStack.Pop();
     } else if ( ((total < 0) && (StatusStack.Top() == NONE))
                   || (start > end) ) {
         // no possible solution with current selections
         --start;
         dummy = StatusStack.Pop();
     }
```

⚠ **WARNING: untested code!**

Iterative KnapSack

```
//Iterative (nonrecursive) solution continued

       else // no soln yet, consider status of current element

          if (StatusStack.Top() == NONE) {
            //try including current array element
            total -= ray[start];
            dummy = StatusStack.Pop();
            StatusStack.Push( INCLUDED );
             ++start;
            StatusStack.Push( NONE );

          } else if (StatusStack.Top() == INCLUDED) {
             // try excluding current array element
             total += ray[start];
            dummy = StatusStack.Pop();
            StatusStack.Push( EXCLUDED );
             ++start;
            StatusStack.Push( NONE );

          } else { // give up on current element & current sum
            --start;
            dummy = StatusStack.Pop();
          }

  } while (!StatusStack.Empty());

  return (found) ;

}//end Knap
```