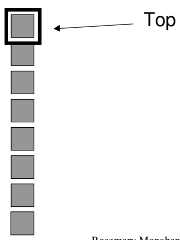
Stacks

SE202 Data Structures

Stacks

- A stack is a special type of list
  - all insertions and deletions take place at one end, called the top
  - thus, the last one added is always the first one available for deletion
  - a dynamic constantly changing object
  - also referred to as
    - pushdown stack
    - pushdown list
    - LIFO list (Last In First Out)

Stacks



Stack Operations

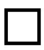
- *Declare*: $\rightarrow$ S :

  The function value of *Declare(S)* is an empty stack

Stack Operations

- *Empty*: $\rightarrow$ S :

  The function *Empty* causes the stack to be emptied and it returns position *End(S)*



Stack Operations

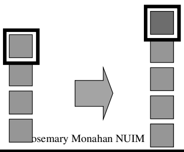
- *IsEmpty*: S $\rightarrow$ B :

  The function value *IsEmpty*(*S*) is *true* if *S* is empty; otherwise it is *false*

Stack Operations

- *Push*: E × S → L :

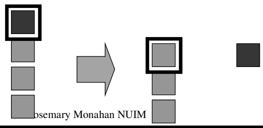
  Push(e, S)
  Insert an element *e* at the top of the stack



Stack Operations

- *Pop*: S → E :

  Pop(S)
  Remove the top element from the stack: i.e. return the top element and delete it from the stack



Stack Operations

- *All these operations can be directly implemented using the LIST ADT operations on a List S*
- *Although it may be more efficient to use a dedicated implementation*
- *It depends what you want: code efficiency or software re-use (i.e. utilization efficiency)*

Array Implementation

```
#include<stdio.h>
#define MAX 20

void push(int); /*prototypes*/
int pop();

int stack[MAX], top = 0;
```


Push

```
// add new data to the top of the stack
void push(int I)
{
 if (top >= MAX)
     {
     printf("The stack is full");
     return;
     }
 else
 {   stack[top] = I;
     top ++;
};
}
```


Pop

```
// remove data from the top of the stack
int pop()
{
 top--;

 if (top < 0)
     {
     printf("The stack is empty\n");
     return 0;
     }
 return(stack[top]);
}
```

Underflow & Overflow

- Stack Underflow
  - occurs when we try to pop the empty stack
- Stack Overflow
  - occurs when we try to push something on the full stack
  - this won't happen in a linked list implementation (unless we run out of memory)

Better Representations of stack?

- Could use
  - a structure:

```
struct stack
{ int items[STACKSIZE];
  int top = 0;
}
```

    with functions pop, push, top,empty …
  - a class with methods pop, push …
  - Data Representation: linked lists, arrays ...

With Structures & Arrays ...

```
void push(stack *ps, int x)
{ if (ps -> top ==STACKSIZE -1)
   { cout << "Stack Overflow";
   }
   else
   { (ps -> top) ++;
     ps -> items[ps -> top] = x;
   }
}
```

Consider using a class and arrays.

What about classes and linked lists?

What do we use stacks for?

- Balanced Expressions
- Converting strings from infix to postfix
- Evaluating postfix expressions
- Storing results in recursive programs
- Removing elements from a list …

Balanced Expressions

- Checking
  - Equal numbers of left and right parentheses
  - Every right parentheses has a matching right
- How?
  - Push opening brackets on the stack
  - Pop matching brackets of the stack
  - If the stack is not empty at the end /we try to pop the empty stack the expression is not balanced

Converting from infix to postfix

- Read in infix string, one item at a time
- If the item is an operand add to the postfix string
- If the item is an operator push it on the stack if

  a) the stack is empty

  b) if the precedence of the item at the top of the stack is < the item being processed
- if the precedence of the item at the top of the stack is > the item being processed pop it to the postfix string
- When the infix string is empty Pop the elements of the stack onto the postfix string to get the result

Evaluating postfix expressions

- Read in string, one item at a time
- If the item is an operand push on the stack
- If the item is an operator pop the stack twice and apply the operand to the resulting values
- Push the result on the stack
- Continue until the string is completed and one value (the answer) remains on the stack.