BBM 201
DATA STRUCTURES

Lecture 5:
Stacks and Queues
Stacks

• A list on which insertion and deletion can be performed.
  • Based on Last-in-First-out (LIFO)

• Stacks are used for a number of applications:
  • Converting a decimal number into binary
  • Program execution
  • Parsing
  • Evaluating postfix expressions
  • Towers of Hanoi
  …
Stacks

A stack is an ordered list in which insertions and deletions are made at one end called the *top*. 
Stacks
Towers of Hanoi

Object of the game is to move all the disks (animals) over to Tower 3. But you cannot place a larger disk onto a smaller disk.
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Stack Operations

1. Pop()
2. Push(x)
3. Top()
4. IsEmpty()

- An insertion (of, say x) is called push operation and removing the most recent element from stack is called pop operation.
- Top returns the element at the top of the stack.
- IsEmpty returns true if the stack is empty, otherwise returns false.

_all of these take constant time - O(1)_
Example

- Push(2)
- Push(10)
- Pop()
- Push(7)
- Push(5)
- Top(): 5
- IsEmpty(): False
Array implementation of stack (pseudocode)

```plaintext
int A[10]
top <- -1  //empty stack

Push(x)
{
    top <- top + 1
    A[top] <- x
}

Pop()
{
    top <- top - 1
}
```

For an empty stack, top is set to -1. In push function, we increment top. In pop, we decrement top by 1.
Array implementation of stack (pseudocode)

```java
Top()
{
    return A[top]
}
IsEmpty()
{
    if(top == -1)
        return true
    else
        return false
}
```

![Array representation of stack](image_url)
Stack
Data Structure

```c
#define MAX_STACK_SIZE 100

typedef struct{
    int VALUE;
}element;

element stack[MAX_STACK_SIZE];
int top=-1;
```
void push (element item)
{
    if(top>=MAX_STACK_SIZE)
    {
        isFull();
        return;
    }
    stack[++top]=item;
}
element pop() {
    if (top == -1)
        return empty_stack();
    return stack[top--];
}
Implementation of Stacks Using Arrays
More array implementation

```c
#include<stdio.h>
#define MAX_SIZE 101
int A[MAX_SIZE];
int top = -1;
void Push(int x) {
    if(top == MAX_SIZE - 1) {
        printf("Error: stack overflow\n");
        return;
    }
    A[++top] = x;
}

void Pop() {
    if(top == -1) {
        printf("Error: No element to pop\n");
        return;
    }
    top--;
}
int Top() {
    return A[top];
}
int main() {
}
```
```c
void Print() {
    int i;
    printf("Stack: ");
    for(i = 0; i <= top; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main() {
    Push(2); Print();
    Push(5); Print();
    Push(10); Print();
    Pop(); Print();
    Push(12); Print();
}```
Check For Balanced Parentheses using Stack

<table>
<thead>
<tr>
<th>Expression</th>
<th>Balanced?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A+B)</td>
<td></td>
</tr>
<tr>
<td>{((A+B)+(C+D))</td>
<td></td>
</tr>
<tr>
<td>{(x+y)*(z)</td>
<td></td>
</tr>
<tr>
<td>[2*3]+(A)</td>
<td></td>
</tr>
<tr>
<td>{a+z}</td>
<td></td>
</tr>
</tbody>
</table>
Check For Balanced Parentheses using Stack

<table>
<thead>
<tr>
<th>Expression</th>
<th>Balanced?</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Yes</td>
</tr>
<tr>
<td>{ ( ) ( ) }</td>
<td>Yes</td>
</tr>
<tr>
<td>{ ( ) ( ) }</td>
<td>No</td>
</tr>
<tr>
<td>[ ] ( )</td>
<td>No</td>
</tr>
<tr>
<td>[ ] ( ) ]</td>
<td>No</td>
</tr>
<tr>
<td>{ }</td>
<td>No</td>
</tr>
</tbody>
</table>

Need: Count of openings = Count of closings
AND
Any parenthesis opened last should be closed first.
Idea: Create an empty list

• Scan from left to right
  
  If opening symbol, add it to the list
  
  Push it into the stack
  
  If closing symbol, remove last opening symbol of the same type
  
  using Pop from the stack

Should end with an empty list
Check For Balanced Parantheses: Pseudocode

CheckBalancedParanthesis(exp)
{
    n← length(exp)
    Create a stack: S
    for i← 0 to n-1
    {
        if exp[i] is '(' or '{' or '['
            Push(exp[i])
        elseif exp[i] is ')’’ or ‘}’’ or ‘]’
        {if (S is not empty)
            (top does not pair with exp[i])
            {return false}
        else
            pop()}
    }
    Return S is empty?

Create a stack of characters and scan this string by using push if the character is an opening parenthesis and by using pop if the character is a closing parenthesis. (See next slide)
Examples

\[
\text{exp} = [(())] \\
i = 2
\]

The pseudo code will return false.

\[
\text{exp} = \{(()())\} \\
i = 5
\]

The pseudo code will return true.
Queues

- A queue is an ordered list on which all insertions take place at one end called the **rear/back** and all deletions take place at the opposite end called the **front**.
- Based on **First-in-First-out (FIFO)**
Comparison of Queue and Stack

Queue ADT

Queue - First-In-First-Out (FIFO)

Stack - Last-In-First-Out (LIFO)
Queue is a list with the restriction that insertion can be made at one end (rear) and deletion can be made at other end (front).
Built-in Operations for Queue

1. Enqueue(x) or Push(x)
2. Dequeue() or Pop()
3. Front(): Returns the element in the front without removing it.
4. IsEmpty(): Returns true or false as an answer.
5. IsFull()

Each operation takes constant time, therefore has $O(1)$ time complexity.
Example

Applications:
- Printer queue
- Process scheduling

Enqueue(2)
Enqueue(5)
Enqueue(3)
Dequeue() \rightarrow 2
Front() \rightarrow 5
IsEmpty() \rightarrow \text{False}
Array implementation of queue (Pseudocode)

```c
int A[10]
front ← -1
rear ← -1
IsEmpty()
  if (front == -1 && rear == -1)
    return true
  else
    return false
Enqueue(x)
  if IsFull()
    return
  elseif IsEmpty()
    front ← rear ← 0
  else
    rear ← rear+1
  A[rear] ← x
```
Array implementation of queue (Pseudocode)

```
Dequeue() {
    if IsEmpty() {
        return
    }
    elseif (front == rear) {
        front ← rear ← -1
    }
    else {
        front ← front+1
    }
}
```

At this stage, we cannot Enqueue an element anymore.
Queue

Data Structure

```c
#define MAX_QUEUE_SIZE 100

typedef struct{
    int value;
}element;

element queue[MAX_QUEUE_SIZE];
int front=-1;
int rear=-1;
```
Add Queue

```c
void addq( element item)  
{
    if(rear==MAX_QUEUE_SIZE-1){
        isFull();
        return;
    }
    queue[++rear]=item;
}
```
Delete Queue

definition

```c
element deleteq(element item)
{
    if(front==rear)
        return isEmpty();
    return queue[++front];
}
```
Circular Queue

- When the queue is full (the rear index equals to MAX_QUEUE_SIZE)
  - We should move the entire queue to the left
  - Recalculate the rear

Shifting an array is time-consuming!
  - $O(\text{MAX_QUEUE_SIZE})$
Circular Queue

- More efficient queue representation
Full Circular Queue

- Front = 0
- Rear = 5

- Front = 4
- Rear = 3
Enqueue for circular array (Pseudocode)

Current position = \( i \)
Next position = \((i+1)\%N\)
previous position = \((i+N-1)\%N\)

**Enqueue(x){**
  
  if \((\text{rear+1})\%N == \text{front}\) return
  
  elseif IsEmpty(){
    front \(\leftarrow\) \text{rear} \(\leftarrow\) 0
  }
  
  else{
    \text{rear} \(\leftarrow\) \((\text{rear+1})\%N\)
    \text{A[rear]} \(\leftarrow\) x
  }

  **Enqueue(15)**
Dequeue for circular array (Pseudocode)

```
Dequeue(x){
  if IsEmpty(){
    return
  } elseif (front == rear){
    front ← rear ← -1
  } else{
    front ← (front+1)%N
  }
```

[Diagram showing a circular array with values and arrows indicating operations.]
Add Circular Queue

```c
void addcircularq(element item)
{
    rear=(rear+1)%MAX_QUEUE_SIZE;
    if(front==rear)
    {
        isFull(rear);
        return;
    }
    queue[rear]=item;
}
```
Delete Circular Queue

```c
void deletecircularq()
{
    if(front==rear)
        return isEmpty();
    front=(front+1)%MAX_QUEUE_SIZE;
    return queue[front];
}
```
References

BBM 201 Notes by Mustafa Ege
- http://www.mycodeschool.com/videos