Lecture 11: Fundamental Data Structures

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Indian Institute of Technology Madras

September 2, 2014

Unordered Array

- We can implement an unordered array in Python using a list

Operations

- `create()`
- `delete()`
- `isEmpty()`
- `length()`
- `find()` *k*-th element
- `search()` for a given element
- `insert()` an element into the list
- `append()`, `join()`, `copy()`, ...

Ordered Array

- Ordered arrays can also be implemented in Python using a list

Operations

- `create()`
- `delete()`
- `isEmpty()`
- `length()`
- `find()` *k*-th element
- `search()` for a given element
- `insert()` an element into the list
- `append()`, `join()`, `copy()`, ...

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(\lg n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(\lg n)$ ...
- In reality, we can get close, and the price must be paid in complexity

How quick do you need your data structure to be!?

Stacks and Queues

- Stacks and queues are more abstract entities than arrays and many other data storage structures
- They're defined primarily by their interface — the permissible operations that can be carried out on them
- The underlying mechanism used to implement them is typically not visible to their users
- The underlying mechanism for a stack can be an array or a linked list

Stack

- Can be implemented using Arrays or Linked Lists

Operations

- `create()`
- `delete()`
- `isEmpty()`
- `push()`
- `pop()`
- `top()`

Delimiter Matching using a Stack

```
from ArrayStack import *

def is_matched(expr):
    '''Return True if all delimiters are properly match;
                                        False otherwise.'''
    lefty = '({[' # opening delimiters
    righty = ')}]' # respective closing delims

    S = ArrayStack()

    for c in expr:
        if c in lefty:
            S.push(c) # push left delimiter on stack
        elif c in righty:
            if S.is_empty():
                return False # nothing to match with
            if righty.index(c) != lefty.index(S.pop()):
                return False # mismatched
    return S.is_empty() # were all symbols matched?
```

Reversing a String using a Stack

```
from ArrayStack import *

def string_reverser(s):
    S = ArrayStack()

    for c in s:
        S.push(c)

    r = []
    while not S.is_empty():
        r.append(S.pop())

    return ''.join(r)

if __name__ == '__main__':
    print(string_reverser('abcdef'))
```

Expression Evaluation using a Stack

Dijkstra's 2-stack algorithm to evaluate fully parenthesised arithmetic expressions:

- Push operands onto the operand stack
- Push operators onto the operator stack
- Ignore left parentheses
- On encountering a right parenthesis, pop an operator, pop the *requisite number of operands*, and push onto the operand stack the result of applying that operator to those operands

Using Stacks to Replace Recursion

- Stacks underlie recursion in computers!

```
from ArrayStack import *

def factorial_stack(n):

    S = ArrayStack()

    while n>1:
        S.push(n)
        n -= 1
    result = 1

    while not S.is_empty():
        result *= S.pop()

    return result

if __name__ == '__main__':
    print(factorial_stack(10))
```

Other Stack Scenarios

- ‘Back button’ on your browser
- Undo stack
- ...