CS 234

Module 3

September 19, 2018

Recipe for provider/plan

Choosing among implementations (provider/plan):

1. Create pseudocode of various options for data structure and algorithms to implement the ADT and its operations.
2. Analyze the costs of each operation for each implementation.
3. Provide options for packages of operation costs.

A brief discussion of memory

A **cell** stores a single bit (0 or 1).

A **block** of memory is a contiguous sequence of cells.

Different data types (e.g. numbers, strings, objects) might require different block sizes. A block might store multiple values in different **fields**.

The location of a cell (or the first cell in a block) is its **address**. A **pointer** is data that is the value of an address.

When a program requests memory for a variable, the operating system finds a block of free memory and associates the name of the variable with its address.

Two main ways of storing multiple pieces of data:

- **Contiguous**: use one block (subdivided)
- **Linked**: use multiple blocks, each with one or more fields containing pointers to other blocks

Data structure: array

An **array** is a block of memory storing multiple **elements**.

The size of the block depends on the number of elements and the number of bits needs to store each element.

To find one of the elements it suffices to have the address/name of the array and the **index**, e.g. `T[i]`.

(Note: This is not the same as the Python data type array.)
Key advantage: An array permits immediate access to any element by **random access**.

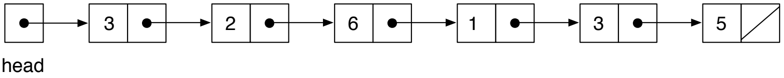
Key disadvantage: Fixed size can lead to running out of space or wasting space.

Data structure: linked list

A **linked list** consists of not-necessarily-contiguous **nodes**, where each node is a block of memory containing data and a pointer to the next node in the linked list.

Key advantage: It is easy to adjust number of nodes or to rearrange parts of the list.

Key disadvantage: Finding a particular node requires following pointers through all preceding nodes.



Types of data structures

Simple structures include:

- arrays
- arrays used with one or more variables
- linked lists
- linked lists used with one or more variables
- linked implementations with pointers in both directions

Complex structures include:

- linked structures where nodes store multiple values and/or multiple pointers
- combinations or arrays and linked lists
- high-level organizations of data that in turn can be implemented using ADTs

Data structure considerations

Keep in mind:

- Different data structures can implement the same ADT.
- Different ADTs can be implemented using the same data structure.
- Different algorithms can be used to implement the same ADT operation on the same data structure.

Customizing data structures:

- Add extra variables.
- Add extra arrays.
- Add extra fields to nodes in a linked implementation (data or pointers).
- In code interfaces, add inputs (e.g. length).

Optimizing choice of data structure:

- Algorithms must always preserve form and meaning of data structure.
- Extra information may be costly to maintain.
- Certain features are best when no additions or deletions occur.

ADT Multiset

The ADT Multiset can store multiple copies of data items.

Preconditions: For all M is a multiset and item any data.

Postconditions: Mutation by Add (add one copy of item) and Delete (delete one copy of item if any are present).

Name	Returns
Create()	a new empty multiset
IsEmpty(M)	true if empty, else false
LookUp(M, item)	true if present, else false
Add(M, item)	
Delete(M, item)	

Figuring out an array implementation

"Wasteful and Stupid array implementation":

- Array of size n, maximum number of additions ever made.
- Variable *next* with the index of the next place to fill.
- Create(): $\Theta(1)$ if uninitialized, $\Theta(n)$ otherwise.
- IsEmpty(M): $\Theta(n)$ to check all.
- LookUp(M, item): $\Theta(n)$ check all indices in worst case.
- Add(M, item) $\Theta(1)$ enter at *next*, update *next*.
- Delete(M, item) $\Theta(n)$ search plus $\Theta(1)$ to delete.

"Better implementation"

- Variable *num* with number of items as well as *next*
- IsEmpty(M): $\Theta(1)$ to check *num*.
- Add(M, item): $\Theta(1)$, update *num* too.
- LookUp(M, item), Delete(M, item): search part reduced to $\Theta(k)$, update *num* too.

Array implementation of ADT Multiset

Provider/plan steps 1-2

Data structure:

- Array of size n with all elements stored contiguously, starting at 0.
- Variable num with the number of items stored.

Worst-case running time of operations as a function of n and k (the number of items stored at the time of the operation):

- Create(): $\Theta(1)$ if uninitialized, $\Theta(n)$ otherwise.
- IsEmpty(M): $\Theta(1)$ - check if num is zero.
- LookUp(M, item): $\Theta(k)$ - check all elements
- Add(M, item): $\Theta(1)$ - enter item at index num, add one to num.
- Delete(M, item): $\Theta(k)$ - search, swap with item at index $num - 1$, subtract one from num.

Note: Every operation must preserve the meaning of the array and variable.

Say

Now give analysis only as a function of n ($\Theta(1)$, $\Theta(1)$, $\Theta(n)$, $\Theta(1)$, $\Theta(n)$).

Linked list implementation of ADT Multiset

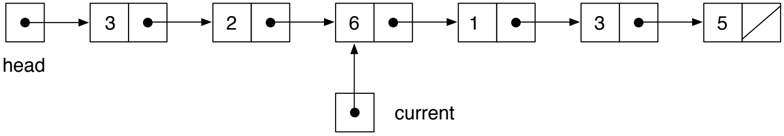
Provider/plan steps 1-2

Data structure:

- Variable *head* pointing to the first node, if any.
- Nodes storing data items and *next* pointers.

Worst-case running time of operations as a function of k (the number of items stored at the time of the operation):

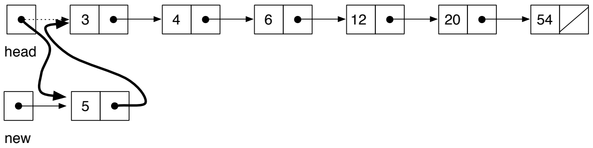
- Create(): $\Theta(1)$ - create variable *head*.
- IsEmpty(M): $\Theta(1)$ - check if *head* points to a node.
- LookUp(M, item): $\Theta(k)$ - check all nodes.

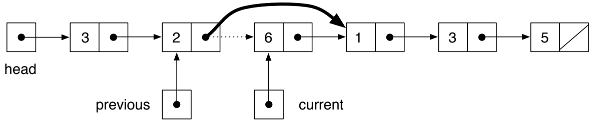


Linked list implementation continued

Worst-case running time of operations as a function of k (the number of items stored at the time of the operation):

- Add(M, item): $\Theta(1)$ - add to front of the list.
- Delete(M, item): $\Theta(k)$ - search, update pointers.





Summary of options

Provider/plan step 3: Provide options for packages of operation costs
Worst-case running time as a function of n (size of the array) and k (the number of items stored at the time of the operation).

	Array	**Linked list**
Create	$\Theta(1)$ or $\Theta(n)$	$\Theta(1)$
IsEmpty	$\Theta(1)$	$\Theta(1)$
LookUp	$\Theta(k)$	$\Theta(k)$
Add	$\Theta(1)$	$\Theta(1)$
Delete	$\Theta(k)$	$\Theta(k)$

Notes:

- User does not need to know name of data structure to make a choice.
- Summary charts will not be provided for future ADTs, but you should consider making your own.
- Not all data structures will have such similar behaviour.

Code interfaces in Python

User/code:

- Agree on the code interface.
- Code a solution to the problem using the ADT.

Provider/code:

- Agree on the code interface.
- Code the chosen data structure and algorithms implementing the ADT.

Classes review:

- __init__ - this can be used for creating an ADT
- self - used in methods to refer to item itself
- __contains__ - can be used for "in", operator overloading
- __eq__ - equals
- __str__ - string representation
- (others)

Design recipe

Previous courses:

- Some steps used in planning
- Some steps evident in comments
- Some steps evident in code

CS 234:

- Choice of ADTs is a new planning step
- Preconditions and postconditions of implementations of operations are handled by contract and purpose of functions.
- See style guide for review of best practices.
- Marks will focus on user/provider division and preconditions and postconditions.
- Use of examples and tests is encouraged for good form and partial marks, but will not be required for marks.

ADT Multiset code interface

User/code and provider/code step 1: Agree on the code interface.

```
class Multiset:
    def __init__(self):
        ''' Multiset() produces a newly
                constructed empty multiset.
        __init__:  None -> Multiset '''

    def empty(self):
        ''' self.empty() produces True if self is empty.
        empty:  Multiset -> Bool '''
```

ADT Multiset code interface continued

```
def __contains__(self, value):
    ''' value in self produces True if
            value is an item in self.
    __contains__:  Multiset Any -> Bool '''

def add(self, value):
    ''' self.add(value) adds value to self.
    Effects:  Mutates self.
    add:  Multiset Any -> None '''

def delete(self, value):
    ''' self.delete(value) removes an item with value
            from self, if any.
    Effects:  Mutates self.
    delete:  Multiset Any -> None '''
```

Coding arrays in Python

Many programming languages have arrays built in. Python doesn't.
Possible approaches:

- Use a Python list instead. (But then can use other list operations.)
- Use the `ctypes` module to access an array. (But uses stuff that you are not expected to understand.)
- Use a class, with a Python list limited to array operations. (Not ideal, but best pedagogically.)

Details of coding arrays in Python

```
class Myarray:
    '''
    Fields: items is a list of items
            size is the size of the array
    '''
    def __init__(self, size):
        '''
        Myarray() produces a newly constructed empty array.
        __init__:  Int -> Myarray
        '''
        self.items = []
        self.size = size
        for index in range(size):
            self.items.append(None)
```

More methods for Myarray

```
def access(self, index):
    '''
    self.access(index) produces the data
            item with given index.
    access:  Myarray, Int -> Any
    Requires:  0 <= index < self.size
    '''
        return self.items[index]


def replace(self, index, item):
    '''
    self.replace(index, item) replaces the data
            item with given index.
    Effects:  Mutates self.
    replace:  Myarray, Int, Any -> None
    Requires:  0 <= index < self.size
    '''
```

Making methods safe

```
def replace(self, index, item):
    '''
    self.replace(index, item) replaces the data
            item with given index or produces
            a warning string.
    Effects:  Mutates self if index in range.
    replace:  Myarray, Int, Any ->
                (anyof "Out of range" None)
    '''
    if index < 0 or index >= self.size:
        return "Out of range"
    else:
        self.items[index] = item
```

Coding Multiset as an array

Provider/code step 2: Code the chosen data structure and algorithms implementing the ADT

Problem: Array size is fixed from the start.

Options:

- Choose a very big size to ensure it will be big enough.
- Choose a small size, and if it gets full, create another bigger array and copy over all the values. Later if too much is not used, create another smaller array and copy it over. (This is what is done for Python lists).
- Change the code interface to ask the user to specify a size.

Coding linked lists in Python

```
class Node:
    '''
    Fields:  item stores any value
            next points to the next node in the list
    '''
    def __init__(self, item, next = None):
        '''
        Node() produces a newly constructed empty node.
        __init__:  None -> Node
        '''
        self.item = item
        self.next = next
```

Coding Multiset as a linked list

Provider/code step 2: Code the chosen data structure and algorithms implementing the ADT

```
from node import *

class Multiset:
    '''
    Field:  _head points to the first node
            in the linked list
    '''
    def __init__(self):
    '''
        Multiset() produces a newly
                constructed empty multiset.
        __init__:  -> Multiset
    '''
        self._head = None
```

Coding Multiset as a linked list, continued

```
def __contains__(self, value):
    '''
    value in self produces True if
            value is an item in self.
    __contains__:  Multiset Any -> Bool
    '''
    current = self._head
    while current != None:
        if current.item == value:
            return True
        else:
            current = current.next
    return False
```

Coding use of ADT Multiset

User/code step 2: Code a solution to the problem using the ADT.

Note: Can use any of the files for implementing Multiset (change after "from").

```
from multisetll import Multiset

data_set = Multiset()
data_file = open("data.txt","r")
data_list = data_file.readlines()
data_file.close()
for line in data_list:
    data_set.add(line.strip())

value = input("Guess a value or type stop:  ")
empty = False
```

Continuation of example

```
while value != "stop" and not empty:
    if value in data_set:
        print("Yes, the set contains", value)
        data_set.delete(value)
    else:
        print("No, the set does not contain", value)
    if data_set.empty():
        print("Sorry, there are no values left to guess.")
        empty = True
    else:
        value = input("Guess a value or type stop:  ")
```

ADT Set

The ADT Set stores at most one copy of each data item.

Preconditions: For all S is a set and item any data.

Postconditions: Mutation by Add (add item if not already present) and Delete (delete item if present).

Name	Returns
Create()	a new empty set
IsEmpty(S)	true if empty, else false
LookUp(S, item)	true if present, else false
Add(S, item)	
Delete(S, item)	

Array implementation of ADT Set

Data structure:

- Array of size *n* with all elements stored contiguously, starting at 0.
- Variable *num* with the number of items stored.

Worst-case running time of operations as a function of *n* and *k* (the number of items stored at the time of the operation):

- Create(): same as Multiset
- IsEmpty(S): same as Multiset
- LookUp(S, item): same as Multiset
- Add(S, item): $\Theta(k)$ - search to see if the item is already stored; if not, then enter at index *num*, add one to *num*.
- Delete(S, item): same as Multiset

Special case: Orderable data

Data is **orderable** if we can compare items not only for equality but also for order.

New data structures:

- Sorted array: items are stored in nondecreasing order.
- Sorted linked list: items are stored in nondecreasing order.

In **successful search** the item is present; in **unsuccessful search** the item is not.

Algorithms:

- Add requires search now. LookUp and Delete still do.
- **Binary search** can be used in a sorted array. The cost is in $\Theta(\log k)$ (k is the number of items stored) in the worst case for both successful and unsuccessful search.
- **Linear search** can be used in a sorted linked list. Worst-case is still in $\Theta(k)$ (k is the number of items stored), but unsuccessful search can stop as soon as a value larger than the search item is found.

Searching as a fundamental operation

In many data structures, the cost of other operations depends on the cost of searching.

Types of algorithms and analysis to consider:

- An algorithm is **comparison-based** if the only types of actions performed on data items are comparisons ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$).
- It is possible to perform actions other than comparisons on **digital data**.
- Data structures may be **internal** (stored entirely in memory) or **external** (making use of external storage).
- When we know (or can guess) something about the frequency with which we search for various data items, we might consider average-case running time instead of worst-case running time.

Adding n items

Worst-case cost of adding n items to a sorted linked list:

- Cost of adding item i is in $\Theta(i)$.
- Total cost is in $\Theta(n^2)$.

Possible improvements:

- User view: Sort items first at cost of $\Theta(n \log n)$, then add in nonincreasing order, for each addition in constant time.
- Provider view: Add a new ADT operation that creates a linked list from a set of values (not necessarily sorted); sort and enter at cost of $\Theta(n \log n)$.

Sorting as a fundamental operation

Sorting and ADTs:

- User view: Use ADT operations to sort data.
- Provider view: Sort data by direct access to data structure.

A sorting algorithm is **stable** if equal-valued items are in the same order before and after sorting.

Revisiting comparison-based sorting algorithms

Selection sort

Sort by repeatedly extracting the smallest remaining value (CS 116).

User view: Like repeatedly using an ADT operation that removes the smallest data item.

Insertion sort

Sort by repeatedly inserting items into a sorted list (CS 116).

Provider view: Like adding to a sorted array or linked list data structure.