CS 234

Module 4

October 3, 2018

Case study

Problem: Given Racket code, are all pairs of parentheses and brackets matched correctly?

Example:

```
(define (is-odd?  x)
  (cond [(= x 0) false]
        [else (is-even?  (sub1 x))]))
```

Recipe for user/plan

1. Determine types of data and operations.
2. For each type, choose/customize an ADT.
3. Develop pseudocode algorithm using ADT operations.
4. Calculate cost of algorithm with respect to costs of operations.
5. Using information from provider, choose best option.

Step 2: Choose/customize an ADT

Idea: Access the "newest" ones first and the "older" ones later.
Support "first in last out" arrangement of data.
Motivation:

- Undo
- Backtrack in a maze
- Balanced parentheses
- Recursion

ADT Stack
Intuition: Like a stack of papers on your desk. Newer papers go on top, and papers are deleted from the top.
Data: Only open parentheses/brackets are used in this application, but in general it can be any data.

ADT Stack

The ADT Stack supports "first in last out" arrangement of data.

Preconditions: For all S is a stack and item any data; for Top and Pop, S is not empty.

Postconditions: Mutation by Push (add item to top of the stack) and Pop (delete item from top of stack).

Name	Returns
Create()	a new empty stack
IsEmpty(S)	true if empty, else false
Top(S)	item that is on top
Push(S, item)	
Pop(S)	item that was on top

Notes:

- Textbook uses peek instead of Top.
- Alternatives for Pop include error if stack is empty or nothing happens if stack is empty.

Example of use of ADT Stack operations

colours $\leftarrow$ Create()
print(IsEmpty(colours))
Push(colours, "puce")
Push(colours, "mauve")
best $\leftarrow$ Top(colours)
Push(colours, "scarlet")
newbest $\leftarrow$ Pop(colours)
print(best)
print(newbest)

Step 3: Pseudocode using ADT operations

Main ideas:

- Push open brackets on stack.
- Everything inside matching brackets should be matched, so when close bracket encountered its match should be on the top of the stack.

What can go wrong:

- Example ([)]: mismatch on top of stack
- Example:))): nothing on top of stack
- Example: (((: extra values on stack at end

Algorithm idea:

- Process brackets in a loop.
- Push open brackets.
- At close bracket, fail if empty or if mismatch.
- When all brackets read, fail if stack is not empty.

Case study pseudocode

```
1   S ← Create()
2   while input is not empty
3       if the next character x is ( or [ or {
4           Push(S,item)
5       else
6           if IsEmpty(S)
7               return "Bad program"
8           y ← Pop(S)
9           if x does not match y
10              return "Bad program"
11  if IsEmpty(S)
12      return "Good program"
13  else
14      return "Bad program"
```

Costs: Create $C(k)$, IsEmpty $E(k)$, Push $A(k)$, Pop $D(k)$

Step 4: Cost of algorithm using cost of operations

Sequential blocks: 1 ($\Theta(C(0))$), 2-10, and 11-14.

Lines 11-14: branching, so cost of 11 plus max of 12 and 14 (total $\Theta(E(k))$)

Lines 2-10, n iterations

- Loop body is branching, 3 ($\Theta(1)$) plus max of 4 ($\Theta(A(k))$) and 6-10
- Lines 6-10: sequential blocks 6-7 ($\Theta(E(k))$), 8 ($\Theta(D(k))$), 9-10 (constant)
- Total for 2-10: $\Theta(n \times \max\{A(k), E(k), D(k)\}$.

Total overall: Max of 1 ($\Theta(C(0))$), 2-10 (above), and 11-14 $\Theta(E(k))$ (dominated by 2-10).
Grand total: $\Theta(C(0)) + n \times \max\{A(k), E(k), D(k)\}$.

Conclusions:

- Can afford more expensive cost of creation since done only once.
- Better to have lower max of others than a few very cheap since max in calculation.

Step 5: Use information from provider

Before Step 5 can take place, the provider needs to work.

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.

Figuring out an array implementation

Unlike in ADT Set and ADT Multiset, we need to store not only data items but information about their position.

Assume we keep items in order.

Key question: Where are the bottom and top of the stack located?

Idea 1: top at index 0
Flip back to example of use of ADT Stack operations.

Idea 2: top at index n-1

Idea 3: bottom at index 0

Array implementation of ADT Stack

Data structure:

- Array of size n with all elements stored contiguously, starting at 0.
- Variable *top* with the last index storing an element (or -1 if empty).

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(): $\Theta(1)$ to set *top* to -1, no initialization required since reads only where elements are stored.
- IsEmpty(S): $\Theta(1)$ - check if *top* is -1.
- Top: $\Theta(1)$ - return item at index *top*.
- Push(S, item): $\Theta(1)$ - increment *top*, store item at index *top*.
- Pop(S): $\Theta(1)$ - remove and return item at index *top*, decrement *top*.

Linked list implementation of ADT Stack

Same three ideas as for stack; try Idea 3 first

Idea 3: bottom in first position

- Can trace through pointers, but $\Theta(k)$ cost.
- Idea of *top* pointer.
- Constant time.

- Ask how to update *top* pointer.
- Consider idea of previous and next.
- Observe that can't update previous.
- $\Theta(k)$ time.

Idea 1: top in first position

Linked list implementation of ADT Stack

Data structure:

- Variable *top* 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 *top*.
- IsEmpty(S): $\Theta(1)$ - check if *top* points to a node.
- Top(S): $\Theta(1)$ - return value in node to which *top* points.
- Push(S, item): $\Theta(1)$ - create new node with *next* pointer set to *top*, update *top* to point to new node.
- Pop(S): $\Theta(1)$ - return value in node to which *top* points, update *top* to the *next* pointer of the node to which it points.

Discussion

For the two implementations, why are the best choices "opposite"? (In terms of where top is stored.)
Intuition:

- Cost depends on lookup and modification.
- For both, choose way to make both cheap.

Looking ahead:

- Stack not useful for students in line for office hours.
- Form ADT that allows adding to end of line, deleting (deleting a student sounds bad) from front of line.
- Notice: need to find and modify both front and back.

ADT Queue

The ADT Queue supports "first in first out" arrangement of data.

Preconditions: For all Q is a queue and item any data; for First and Dequeue, Q is not empty.

Postconditions: Mutation by Enqueue (add item to back of the queue) and Dequeue (delete item from front of queue).

Name	Returns
Create()	a new empty queue
IsEmpty(Q)	true if empty, else false
First(Q)	item that is first
Enqueue(Q, item)	
Dequeue(Q)	item that was first

Example of use of ADT Queue operations

animals ← Create()
print(IsEmpty(animals))
Enqueue(animals, "dingo")
Enqueue(animals, "echidna")
best ← First(animals)
Dequeue(animals, "wombat")
newbest ← Dequeue(animals)
print(best)
print(newbest)

Data structures for ADT Queue

Ask:

- How to speed up lookup of front and back? (Answer: variables)
- How to "reuse" space at front? (Answer: allow "wrapping")

Circular array with front and back marked.
Discuss $\Theta(1)$ operations, briefly.
Linked list:

- Keep track of front and back of list.
- Delete at front and reconnect.
- Add at back.
- Recall from stack that push was OK when top at the end, the problem was just pop - not done here.
- $\Theta(1)$ time operations.

Other linked data structures

Doubly-linked list

- Nodes have pointers to previous and next.
- Can have pointers to first and last.

Circular list

- Last node points to first node.
- Can have pointer to last (then first easy to find)

Doubly-linked circular list

- Nodes have pointers to previous and next.
- Last node points to first node.
- Can have pointers to first or last (other easy to find)

ADTs with order imposed by operations

ADTs to consider:

- "First in last out" - ADT Stack
- "First in first out" - ADT Queue
- Specify a particular position - ADT List (not same as Python data type list)
- Specify a ranking - ADT Ranking

Customizing ADTs:

- Support additional operations
- Support a different set of operations
- Allow extra dimensions or positions - ADT Grid

ADT List

The ADT List allows access to data items by index.

Preconditions: For all L is a list, pos an integer ≥ 0, item any data.

Postconditions: Mutation by Add (add/replace item at index) and Delete (delete item at pos, if any).

Name	Returns
Create()	a new empty list
IsEmpty(L)	true if empty, else false
Last(L)	last position storing an item
LookUp(L, pos)	item with given pos, else false
Add(L, item, pos)	
Delete(L, pos)	

Example of use of ADT List operations

gifts $\leftarrow$ Create()
print(IsEmpty(gifts))
Add(gifts, "rings", 5)
Add(gifts, "swans", 7)
print(Last(gifts))
Add(gifts, "cows", 2)
fifth $\leftarrow$ LookUp(gifts, 5)
Add(gifts, "doves", 2)
second $\leftarrow$ LookUp(gifts, 2)
print(fifth)
print(second)

Possible array implementations of ADT List

Idea 1: Two arrays, one with positions and one with values, plus variable *last*

- All nonempty array entries first.
- IsEmpty $\Theta(1)$
- Use variable *last* Last. $\Theta(1)$
- For LookUp, Delete, and Add (to check if item with search position is present), find index of search position in position array, and then find the value at the same index in the value array. Also update *last*. $\Theta(k)$ for k values stored.
- For Delete, also need to adjust so that all elements are contiguous, starting at 0.

Idea 2: Customized array with (pos, value) pairs, plus variable *last*

Customizing array: Extra field in each node so that both pos and value stored.

- Costs like in Idea 1.
- Without the variable, both 1 and 2 have $\Theta(k)$ cost for Last.

Array implementation of ADT List

Data structure:

- Array of size n for $n-1$ maximum position; index in array matches position.
- Variable *last* with the last index storing an element (or -1 if empty).

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(): $\Theta(n)$ - initialize array
- IsEmpty(L): $\Theta(1)$ - check if *last* is -1
- Last(L): $\Theta(1)$ - return value in *last*
- LookUp(L, pos): $\Theta(1)$ - return element at index position
- Add(L, item, pos): $\Theta(1)$ - add/replace item at index pos, update *last* if needed.
- Delete(L, pos):$\Theta(1)$ - replace item at index pos with empty, update *last* if needed.

Linked implementations of ADT List

Idea 1: node in linked list position i for pos i (some empty)

Idea 2: (pos, value) pairs in any order

Idea 3: (pos, value) pairs in order by index

Idea 4: doubly-linked list with (pos, value) pairs in order by pos

Variants on LookUp:

- Can store max and min values of pos stored, or use length to choose which direction - then search from front or back.
- Search in alterating steps from front to back.

Observations

About list implementations:

- A list (ADT) is not an array (data structure).
- A list does not have to be implemented as an array.
- An array implementation of a list does not have to be the obvious one.

About data structures:

- A data structure can consist of multiple arrays.
- An array can store more than one piece of information at an index.
- A linked list can store more than one piece of information in a node.

Cost of ADT List LookUp(L,pos)

Given function of k, the number of values stored

- array, worst case $\Theta(1)$
- array, best case $\Theta(1)$
- linked list, worst case $\Theta(k)$ (last pos)
- linked list, best case $\Theta(1)$ (first pos)

Given function of k, the number of values stored, and p, the value of the pos

- array, worst case $\Theta(1)$
- array, best case $\Theta(1)$
- linked list, worst case $\Theta(p)$
- linked list, best case $\Theta(p)$

ADT Ranking

The ADT Ranking supports a **ranking** of data items (an assignment of distinct ranks from 0 to $k-1$ for k data items).

Preconditions: For all R is a ranking, item any data; for LookUp and Delete r is the rank of an element in R, and for Add r can be the rank of an element in R or one greater.

Postconditions: Mutation by Add (item has rank r, all items that were rank r or greater have their ranks incremented) and Delete (item at rank r removed, all items that were at rank r+1 or greater have their ranks decremented).

Name	Returns
Create()	a new empty ranking
IsEmpty(R)	true if empty, else false
MaxRanking(R)	maximum rank used
LookUp(R, r)	item with ranking r
Add(R, item, r)	
Delete(R, r)	item that had ranking r

Example of use of ADT Ranking operations

desserts ← Create()
print(IsEmpty(desserts))
Add(desserts, "cake", 0)
Add(desserts, "pie", 0)
Add(desserts, "cookies", 2)
best ← LookUp(desserts, 0)
Add(desserts, "ice cream", 1)
maybe ← LookUp(desserts, 2)
print(best
print(maybe)
print(MaxRanking(desserts))

Data structures for ADT ranking

Idea 1: Array with (rank, value) pairs, unordered

- LookUp, Add, Delete: all entries searched, for add and delete may need to update ranks on all

Idea 2: Array with item of rank r at index r

- Worst case cost of Add and Delete $\Theta(k)$ due to moving elements.
- Best case cost of these $\Theta(1)$ for last ranked item.
- All other operations worst-case cost $\Theta(1)$.

Idea 3: linked list, position is rank

- Cost of search plus constant time modification.
- Cost of $\Theta(k)$ as a function of k, or $\Theta(r)$ as a function of k and r for r the rank.

Idea 4: Doubly-linked list, position is rank

If a function of k and r, then $\Theta(\min(r, k-r))$ to search, choosing from front or back (with $\Theta(1)$ to modify).

Customizing ADTs

Adding operations to ADTs:

- Given a set of data, create an ADT storing all of that data.
- Support a variety of operations from multiple ADTs (e.g. Stack and Queue).
- Support operations typical for the Python data type list, such as reverse and slice.

Note: More operations supported may lead to higher running times for all operations.

Note: Consider special cases where some operations are removed if never needed, such as if data never changes.

Adding dimensions to ADTs:

- Generalize lists to grids (e.g. for game boards or images) or to higher dimensions.

ADT Grid

The ADT Grid allows access to data items by two indices.
Preconditions: For all G is a grid, x and y integers ≥ 0, item any data.

Postconditions: Mutation by Add (add/replace item at (x, y)) and Delete (delete item at (x, y), if any).

Name	Returns
Create()	a new empty grid
IsEmpty(G)	true if empty, else false
LookUp(G, x, y),	item at (x, y), else false
Add(G, item, x, y)	
Delete(G, x, y)	

Data structures for ADT Grid

Consider using ADT List for the first dimension and then ADT List again for the second dimension.
Choose array or linked for each one.

Key new idea

ADTs can be used in the formation of data structures. Here the provider is the user of the ADT, with a new provider being needed to implement the ADT.