CS 234: Data Types and Structures

Naomi Nishimura

Module 4

Date of this version: October 1, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

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/modify/create 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.

ADT Stack

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

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

Postconditions: Mutation by *Push* (adds *Data* to the top of the stack) and *Pop* (deletes data item from the top of the stack).

Name	Returns
Create()	a new empty stack
Is_Empty(S)	*True* if empty, else *False*
Top(S)	data item that is on top
Push(S, Data)	
Pop(S)	data item that was on top

Example of use of ADT Stack operations

```
$Colours \leftarrow$ CREATE()
print(IS_EMPTY(Colours))
PUSH(Colours, "puce")
PUSH(Colours, "mauve")
$One \leftarrow$ TOP(Colours)
PUSH(Colours, "scarlet")
$Two \leftarrow$ POP(Colours)
print(One)
print(Two)
```

Case study pseudocode

```
Symbols ← CREATE()
for each character Char in the input string
    if Char is ( or [ or {
        PUSH(Symbols, Char)
    if Char is ) or ] or }
        if IS_EMPTY(Symbols)
            return "Bad program"
        Current ← POP(Symbols)
        if Current is not the same type of symbol as Char
            return "Bad program"
if IS_EMPTY(Symbols)
    return "Good program"
else
    return "Bad program"
```

Costs: *CREATE* C, *IS_EMPTY* $E(k)$, *PUSH* $A(k)$, *POP* $D(k)$

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 structures 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.

Contiguous implementation of ADT Stack

Data structures:

- Array of size *s* with all data items stored contiguously, starting at index 0
- Variable *Top* storing index of last slot storing a data item (or -1 if empty)

Algorithms:

- *CREATE()* does not initialize array; *Top* set to -1
- *IS_EMPTY(S)* checks if *Top* is -1
- *TOP(S)* returns item at index *Top*
- *PUSH(S, Data)* increments *Top*; stores *Data* at index *Top*
- *POP(S)* removes and returns data item at index *Top*; decrements *Top*

Analysis of contiguous implementation

- *CREATE()*: $\Theta(1)$
- *IS_EMPTY(S)*: $\Theta(1)$
- *TOP(S)*: $\Theta(1)$
- *PUSH(S, Data)*: $\Theta(1)$
- *POP(S)*: $\Theta(1)$

Linked list implementation of ADT Stack

Data structures:

- Variable *Top* storing a pointer to a linked list of nodes, each storing a data item and a pointer to the next node in the list

Algorithms:

- *CREATE()* sets variable *Top* to null pointer
- *IS_EMPTY(S)* checks if *Top* is null
- *TOP(S)* returns the value in the node to which *Top* points
- *PUSH(S, Data)* creates a new node storing *Data* and with its *Next* pointer set to *Top*; updates *Top* to point to the new node
- *POP(S)* returns the value in the node to which *Top* points; updates *Top* to the *Next* pointer of the node to which it points

Analysis of linked implementation

- *Create()*: $\Theta(1)$
- *Is_Empty(S)*: $\Theta(1)$
- *Top(S)*: $\Theta(1)$
- *Push(S, Data)*: $\Theta(1)$
- *Pop(S)*: $\Theta(1)$

ADT Queue

The ADT Queue supports “first in first out” arrangement of data.

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

Postconditions: Mutation by *Enqueue* (adds *Data* to the back of the queue) and *Dequeue* (deletes data item from the front of the queue).

Name	Returns
Create()	a new empty queue
Is_Empty(Q)	*True* if empty, else *False*
First(Q)	data item that is first
Enqueue(Q, Data)	
Dequeue(Q)	data item that was first

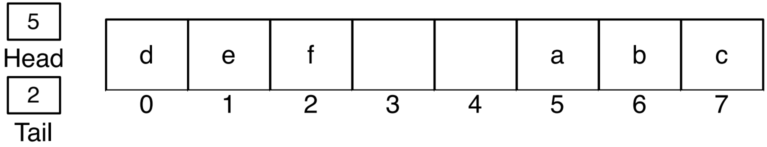
Example of use of ADT Queue operations

```
$Animals \leftarrow$ CREATE()
print(IS_EMPTY(Animals))
ENQUEUE(Animals, "dingo")
ENQUEUE(Animals, "echidna")
$One \leftarrow$ DEQUEUE(Animals)
ENQUEUE(Animals, "wombat")
$Two \leftarrow$ DEQUEUE(Animals)
print(One)
print(Two)
```

Data structure: Circular array

Contiguous:

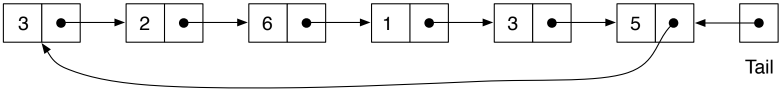
- Array with entries
- *Head* stores index of the first item in the sequence
- *Tail* stores the last item stored in the sequence, or -1 if the sequence is empty



Data structure: Circular linked list

Linked structure:

- Each linked node contains a data item and a *Next* pointer
- *Next* pointer of last node points to first node
- *Tail* pointer to last node

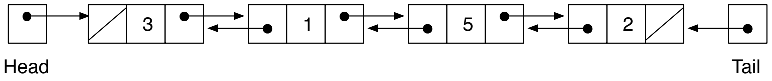


Note: Both the first and the last data items can be found in constant time.

Data structure: Doubly-linked list

Linked structure:

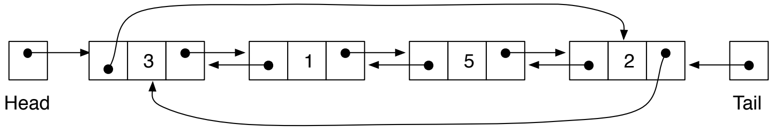
- Each linked node contains a data item, a *Next* pointer, and a *Prev* pointer
- *Head* pointer to first node
- Optional *Tail* pointer to last node



Data structure: Doubly-linked circular list

Linked structure:

- Each linked node contains a data item, a *Next* pointer, and a *Prev* pointer
- *Next* pointer of last node points to first node
- *Prev* pointer of first node points to last node
- Either one or both of *Head* pointer and *Tail* pointer are included



Note: Both the first and the last data items can be found in constant time.

ADTs with order imposed by operations

ADTs to consider:

- “First in last out” - ADT Stack
- “First in first out” - ADT Queue
- Specify a position in one dimension - ADT Indexed Sequence
- Specify a ranking - ADT Ranking
- Specify a position in two dimensions - ADT Grid

Mimicking behaviour of contiguous memory

Without specifying how the data is actually stored, create an ADT with the same features as an array:

- Specify the capacity of the ADT at the outset.
- Allow random access of data items by index.

Notes:

- Positions of data items depend on operations, not values
- Data items need not be orderable

ADT Indexed Sequence

The ADT Indexed Sequence allows access to data items by index.

Preconditions: For all *I* is an indexed sequence, *Data* is any data item, *Cap* is a positive integer; and *Index* is an integer, $0 \leq Index < Cap$.

Postconditions: Mutation by *ADD* (adds/replaces item at *Index*) and *DELETE* (deletes data item at *Index*, if any).

Name	Returns
CREATE(Cap)	a new empty indexed sequence of capacity *Cap* with all entries initialized to empty
CAP(I)	the capacity of *I*
IS_EMPTY(I)	*True* if empty, else *False*
LOOK_UP(I, Index)	data item with index *Index*, if any, else *False*
ADD(I, Index, Data)	
DELETE(I, Index)	data item with index *Index*, if any, else *False*

Note: We assume that *False* is not a data item.

Example of use of ADT Indexed Sequence operations

```
$Gifts \leftarrow$ CREATE(12)
print(CAP(Gifts))
ADD(Gifts, 2, "cows")
ADD(Gifts, 5, "rings")
ADD(Gifts, 7, "swans")
print(IS_EMPTY(Gifts))
$Discarded \leftarrow$ DELETE(Gifts, 5)
$Fifth \leftarrow$ LOOK_UP(Gifts, 5)
ADD(Gifts, 2, "doves")
$Second \leftarrow$ LOOK_UP(Gifts, 2)
print(Fifth)
print(Second)
```

Contiguous implementation 1

Idea: Have one array of indices and one of values.

Data structures:

- Array *Indices* of size *s* of indices (full slots first, any order)
- Array *Values* of size *s* of values (match order in *Indices*)
- Variable *Cap* storing *s*
- Variable *Last* containing index of last full entry or -1

Contiguous implementation 2

Idea: Store the index and value in an ADT Pair with operations *Index* and *Value* that return the two pieces of data.

Data structures:

- Array *Pairs* of size *s* storing ADT Pairs
- Variable *Cap* storing *s*
- Variable *Last* containing index of last full entry or -1

Contiguous implementation 3

Data structures:

- Array of size *s*; index in array matches position
- Variable *Cap* storing *s*

Algorithms:

- *CREATE* initializes all entries to empty; sets *Cap* to *s*
- *CAP* produces *Cap*
- *IS_EMPTY* checks entries until all checked or a value is found
- *LOOK_UP* checks position *Index* in array
- *ADD* updates value at position *Index* in array
- *DELETE* sets entry at position *Index* to empty

Linked implementations of ADT Indexed Sequence

Ideas for linked lists (also consider other data structures):

1. *s* nodes, where the node in position *Index* stores the data item for *Index*
2. Each node stores an ADT Pair, any order
3. Each node stores an ADT Pair, ordered by index
4. Each node stores an ADT Pair, ordered by value

Observations

About implementations:

- An indexed sequence (ADT) is not an array (data structure).
- An indexed sequence does not have to be implemented as an array.
- A contiguous implementation of an indexed sequence 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 linked node.

Cost of ADT Indexed Sequence *Look_Up(I,Index)*

As a function of s:

- Contiguous 3, worst case $\Theta(1)$
- Contiguous 3, best case $\Theta(1)$
- Linked with s nodes, worst case $\Theta(s)$ (last position)
- Linked with s nodes, best case $\Theta(1)$ (first position)

As a function of s and p, the value of *Index*:

- Contiguous 3, worst case $\Theta(1)$
- Contiguous 3, best case $\Theta(1)$
- Linked with s nodes, worst case $\Theta(p)$
- Linked with s nodes, best case $\Theta(p)$

Mimicking behaviour of linked memory

Without specifying how the data is actually stored, create an ADT with the same features as a linked list:

- Each data item has a rank based on its position in the linked list.
- Adding or deleting an item changes the ranks of other items.

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, *Data* any data item; for *LOOK_UP* and *DELETE* *Rank* is the rank of a data item in *R*, and for *ADD* *Rank* can be the rank of a data item in *R* or one greater.

Postconditions: Mutation by *ADD* (adds *Data* with rank *Rank* and increments ranks of all other items with rank *Rank* or greater) and *DELETE* (deletes data item with rank *Rank* and decrements ranks of all items that had rank *Rank*+1 or greater).

Name	Returns
CREATE()	a new empty ranking
IS_EMPTY(R)	*True* if empty, else *False*
MAX_RANKING(R)	maximum rank used (-1 if empty)
LOOK_UP(R, Rank)	data item that has rank *Rank*
ADD(R, Rank, Data)	
DELETE(R, Rank)	data item that had rank *Rank*

Example of use of ADT Ranking operations

```
Desserts ← CREATE()
print(IS_EMPTY(Desserts))
ADD(Desserts, 0, "cake")
ADD(Desserts, 0, "pie")
ADD(Desserts, 2, "cookies")
One ← LOOK_UP(Desserts, 0)
ADD(Desserts, 1, "ice cream")
Two ← LOOK_UP(Desserts, 2)
print(One)
print(Two)
print(MAX_RANKING(Desserts))
```

Data structures for ADT Ranking

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

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

Idea 3: Linked list, position is rank

Idea 4: Doubly-linked list, position is rank

ADT Grid

The ADT Grid allows access to data items by two indices.
Preconditions: For all *G* is a grid, *Num_Rows*, *Num_Cols*, *Row*, and *Col* are nonnegative integers, $0 \leq Row < Num\_Rows$, $0 \leq Col < Num\_Cols$, *Data* any data item.

Postconditions: Mutation by *ADD* (adds/replaces item at (*Row*, *Col*)) and *DELETE* (deletes data item at (*Row*, *Col*), if any).

Name	Returns
CREATE(Num_Rows, Num_Cols)	a new grid of dimensions *Num_Rows* × *Num_Cols* with all entries initialized to empty
ROWS(G)	number of rows
COLUMNS(G)	number of columns
IS_EMPTY(G)	*True* if empty, else *False*
LOOK_UP(G, Row, Col)	data item at *(Row, Col)*, if any, else *False*
ADD(G, Row, Col, Data)	
DELETE(G, Row, Col)	data item at *(Row, Col)*, if any, else *False*

Data structures for ADT Grid

Consider using ADT Indexed Sequence for the first dimension and then ADT Indexed Sequence again for the second dimension.

The provider for the ADT Grid is now the user for the ADT Indexed Sequence.

Key new idea

ADTs can be used in implementations.

Module summary

Topics covered:

- Case study: Racket code
- ADT Stack
- ADT Queue
- Data structure: Circular array
- Data structure: Circular linked list
- Data structure: Doubly-linked list
- Data structure: Doubly-linked circular list
- ADT Indexed Sequence
- ADT Ranking
- ADT Grid