ECE 2400 Computer Systems Programming
Fall 2017

Topic 9: Abstract Data Types

School of Electrical and Computer Engineering
Cornell University

revision: 2017-10-11-11-58

- We discussed importance of separating interface from abstraction
- An abstract data type (ADT) is the definition of just the interface for a data structure: specifies just *what* the data structure does, not *how* the data structure does it
- The list and vector data structures were a good first step towards true ADTs, although these are usually regarded as "low-level" ADTs and often are used to implement "high-level" ADTs
- In this topic, we will discuss the *interface*, *implementation*, and *uses* of five "high-level" ADTs
  - Sequences begin, end, next, insert, remove, get, set
  - Stacks push, pop, empty
  - Queues enq, deq, empty
  - Sets add, remove, contains, union, intersect
  - Maps add, remove, lookup

1. Sequence ADTs

- Imagine putting together a music playlist
- We can insert songs into any position in the playlist
- We can remove songs from any position in the playlist
- We can access/change (get/set) songs anywhere in the playlist
- We can iterate through the playlist to play the music

Sequence ADT interface

- Pseudocode for working with a sequence ADT

```
insert 2 at beginning of sequence
insert 4 at end of sequence
insert 6 at end of sequence
insert 3 at beginning of sequence
set iterator to beginning of sequence
while iterator is not equal to end of sequence
  get value at iterator
  set iterator to next iterator
```

- C-based interface for sequence ADT

```
typedef struct
{
  // opaque
}
seq_t;
```

```
typedef /* opaque */       itr_t;
typedef /* user defined */ item_t;
```

```
void   seq_construct ( seq_t* seq );
void   seq_destruct  ( seq_t* seq );
itr_t  seq_begin     ( seq_t* seq );
itr_t  seq_end       ( seq_t* seq );
itr_t  seq_next      ( seq_t* seq, itr_t itr );
void   seq_insert    ( seq_t* seq, itr_t itr, item_t v );
void   seq_remove    ( seq_t* seq, itr_t itr );
item_t seq_get       ( seq_t* seq, itr_t itr );
void   seq_set       ( seq_t* seq, itr_t itr, item_t v );
```

```
1  int main( void )
2  {
3    seq_t seq;
4    seq_construct ( &seq );
5    seq_insert    ( &seq, seq_begin(&seq), 2 );
6    seq_insert    ( &seq, seq_end  (&seq), 4 );
7    seq_insert    ( &seq, seq_end  (&seq), 6 );
8    seq_insert    ( &seq, seq_begin(&seq), 3 );
9
10   itr_t itr = seq_begin( &seq );
11   while ( itr != seq_end( &seq ) )
12   {
13     int value = seq_get( &seq, itr );
14     itr = seq_next( &seq, itr );
15   }
16
17   seq_destruct  ( &seq );
18   return 0;
19 }
```

Sequence ADT implementations

- List implementation
  - dynamically allocated nodes and pointers
  - `itr_t` is a pointer to a node
  - `seq_begin` returns the head pointer
  - `seq_end` returns the NULL pointer
  - `seq_next` returns `itr->next_p`
- Vector implementation
  - dynamically allocated array (with resizing)
  - `itr_t` is an index
  - `seq_begin` returns 0
  - `seq_end` returns size
  - `seq_next` returns `itr++`

- Worst case time complexity for sequence ADT implementations

	`begin`	`end`	`next`	`insert`	`remove`	`get`	`set`
list	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vector	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(1)$

2. Stack ADTs

- Imagine a stack of playing cards
- We can add (push) cards onto the top of the stack
- We can remove (pop) cards from the top of the stack
- Not allowed to insert cards into the middle of the deck
- Only the top of the stack is accessible
- Sometimes called last-in, first-out (LIFO)

Stack Interface

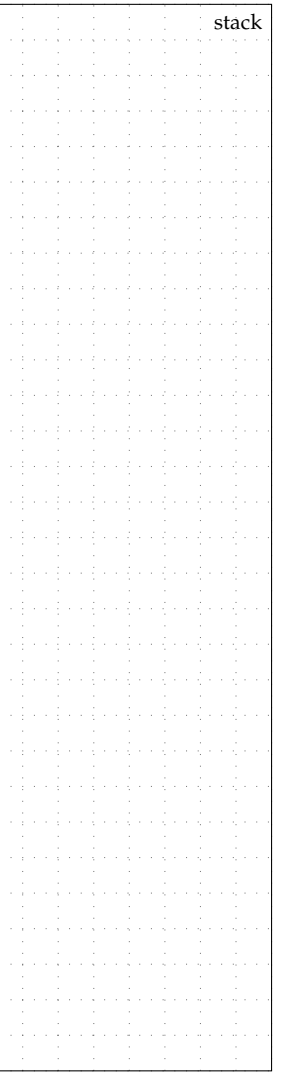
- Pseudocode for working with a stack ADT

```
push 6 onto stack
push 2 onto stack
pop    from stack
push 8 onto stack
push 3 onto stack
pop    from stack
pop    from stack
pop    from stack
```

- C-based interface for stack ADT

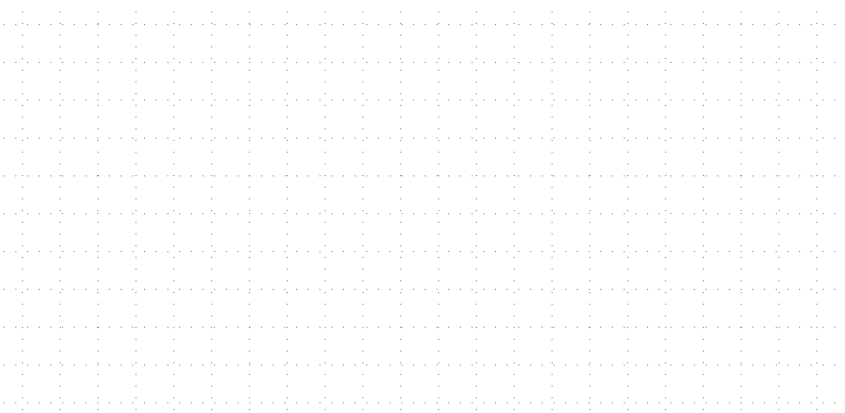
```
typedef struct
{
  // opaque
}
stack_t;

typedef /* user defined */ item_t;

void   stack_construct ( stack_t* stack_p );
void   stack_destruct  ( stack_t* stack_p );
void   stack_push      ( stack_t* stack_p, item_t v );
item_t stack_pop       ( stack_t* stack_p );
int    stack_empty     ( stack_t* stack_p );
```

```
1  int main( void )
2  {
3    stack_t stack;
4    stack_construct   ( &stack );
5    stack_push        ( &stack, 6 );
6    stack_push        ( &stack, 2 );
7    // Stack now has two items
8
9    int a = stack_pop ( &stack );
10   stack_push        ( &stack, 8 );
11   stack_push        ( &stack, 3 );
12   // Stack now has three items
13
14   int b = stack_pop ( &stack );
15   int c = stack_pop ( &stack );
16   int d = stack_pop ( &stack );
17   // Stack is now empty
18
19   stack_destruct    ( &stack );
20   return 0;
21 }
```



Stack Implementation

- List implementation
  - `stack_push/stack_pop` operate on tail of list (`list_push_back`)
  - `stack_pop` should be straight-forward to implement (`list_pop_back`)
- Vector implementation
  - `stack_push/stack_pop` operate on end of vector (`vector_push_back`)
  - `stack_pop` should be straight-forward to implement (`vector_pop_back`)
  - What is worst case time complexity for `stack_push`?



- Worst case time complexity for stack ADT implementations

	`push`	`pop`	`empty`
list	$O(1)$	$O(1)$	$O(1)$
vector	$O(1)$*	$O(1)$	$O(1)$

(*) amortized

Stack Uses

- Parsing HTML document, need to track currently open tags

```
<html>
<head>
<title>Simple Webpage</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
</i> just bold</b>
</body>
</html>
```

- Undo log in text editor or drawing program
  - After each change push entire state of document on stack
  - Undo simply pops most recent state of document off of stack
  - Redo can be supported with a second stack
  - When popping a state from undo stack, push that state onto redo stack

3. Queue ADTs

- Imagine a queue of people waiting for coffee at College Town Bagels
- People enqueue (enq) at the back of the line to wait
- People dequeue (deq) at the front of the line to get coffee
- People are not allowed to cut in line
- Sometimes called first-in, first-out (FIFO)

Queue Interface

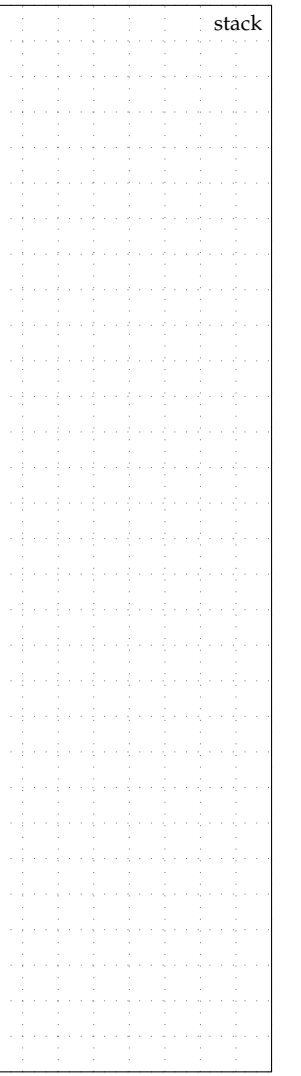
- Pseudocode for working with a queue ADT

```
enq  6 onto tail of queue
enq  2 onto tail of queue
deq    from head of queue
enq  8 onto tail of queue
enq  3 onto tail of queue
deq    from head of queue
deq    from head of queue
deq    from head of queue
```

- C-based interface for queue ADT

```
1  typedef struct
2  {
3    // opaque
4  }
5  queue_t;
6
7  typedef /* user defined */ item_t;
8
9  void   queue_construct ( queue_t* queue_p );
10 void   queue_destruct  ( queue_t* queue_p );
11 void   queue_enq       ( queue_t* queue_p, item_t v );
12 item_t queue_deq       ( queue_t* queue_p );
13 int    queue_empty     ( queue_t* queue_p );
```

```
1  int main( void )
2  {
3    queue_t queue;
4    queue_construct   ( &queue );
5    queue_enq         ( &queue, 6 );
6    queue_enq         ( &queue, 2 );
7    // Queue now has two items
8
9    int a = queue_deq ( &queue );
10   queue_enq         ( &queue, 8 );
11   queue_enq         ( &queue, 3 );
12   // Queue now has three items
13
14   int b = queue_deq ( &queue );
15   int c = queue_deq ( &queue );
16   int d = queue_deq ( &queue );
17   // Queue is now empty
18
19   queue_destruct    ( &queue );
20   return 0;
21 }
```



Queue Implementation

- List implementation
  - `queue_enq` operates on tail of list (`list_push_back`)
  - `queue_deq` operates on head of list (`list_pop_front`)
- Vector implementation
  - `queue_enq` operates on end of vector (`vector_push_back`)
  - `queue_deq` operates on beginning of vector (`vector_pop_front`)
  - `queue_deq` requires copying all items

- Circular buffer implementation
  - Keep head and tail indices
  - `queue_enq` inserts item at tail index and increments tail index
  - `queue_deq` removes item at head index and increments head index
  - Indices are always incremented so that they "wrap around" buffer
  - Can dynamically resize just like in the vector

- Worst case time complexity for queue ADT implementations

	`enq`	`deq`	`empty`
list	$O(1)$	$O(1)$	$O(1)$
vector	$O(1)$*	$O(N)$	$O(1)$
circular buffer	$O(1)$*	$O(1)$	$O(1)$

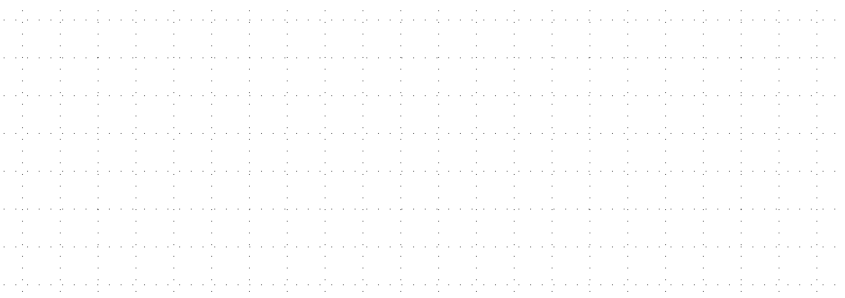
(*) amortized

Queue Uses

- Network processing
  - Operating system provides queues for NIC to use
  - Each network request is enqueued into the queue
  - Operating system dequeues and processes these requests in order
- Some algorithms process work item, generate new work items
  - Algorithm dequeues work item ...
  - ... processes work item and enqueues new work items
  - Algorithm repeats until queue is empty

4. Set ADTs

- Imagine we are shopping at Greenstar with a friend
- Both of have our own shopping bags
- As I go through the store, I add items to my shopping bag
- I might also remove items from my shopping bag
- I might need to see if my bag already contains an item
- We might want to see if we both have the same item (intersection)
- We might want to combine our bags before we checkout (union)
- We don't care about the order of items in the bag



Set Interface

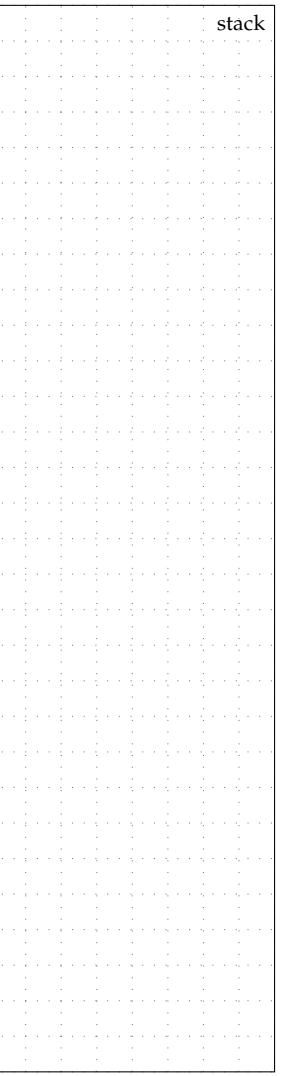
- Pseudocode for working with a set ADT

```
add    2 to   set0
add    4 to   set0
add    6 to   set0
does set0 contain 4?
add    6 to   set1
add    5 to   set1
set set2 to union of set0 and set1
```

- C-based interface for set ADT

```
typedef struct
{
  // opaque
}
set_t;

typedef /* user defined */ item_t;

void set_construct ( set_t* set );
void set_destruct  ( set_t* set );
void set_add       ( set_t* set, item_t v );
void set_remove    ( set_t* set, item_t v );
int  set_contains  ( set_t* set, item_t v );

void set_intersect ( set_t* set_dest,
                     set_t* set_src0, set_t* set_src1 );

void set_union     ( set_t* set_dest,
                     set_t* set_src0, set_t* set_src1 );
```

```
1  int main( void )
2  {
3    set_t set0;
4    set_construct ( &set0 );
5    set_add       ( &set0, 2 );
6    set_add       ( &set0, 4 );
7    set_add       ( &set0, 6 );
8
9    if ( set_contains( &set0, 4 ) ) {
10     ... more code ...
11   }
12
13   set_t set1;
14   set_construct ( &set1 );
15   set_add       ( &set1, 4 );
16   set_add       ( &set1, 6 );
17
18   set_t set3;
19   set_union ( &set3, &set0, &set1 );
20
21   set_destruct  ( &set0 );
22   set_destruct  ( &set1 );
23   set_destruct  ( &set2 );
24   return 0;
25 }
```



Set Implementation

- List implementation
  - `set_add` need to search list first ...
  - ... if not in list then add to end of list (`list_push_back`)
  - `set_remove` needs to search list
  - `set_contains` needs to search list
  - `set_intersection` for each element in one list, search other list
  - `set_union` needs to iterate over both input lists
- Vector implementation
  - `set_add` need to search list first ...
  - ... if not in list then add to end of vector (`vector_push_back`)
  - `set_remove` needs to search vector, shift elements over
  - `set_contains` needs to search vector
  - `set_intersection` for each element in one vector, search other list
  - `set_union` needs to iterate over both input lists
- Lookup Table Implementation
  - Use a vector which is indexed by the value we want to store in set
  - Element is zero if the corresponding value is not in set
  - Element is one if the corresponding value is in set
  - Only possible if values in set can be transformed into integers
  - Can be efficient if range of possible values in set are small

- Bit Vector Implementation
  - Use one or more `ints`
  - Each bit position represents a value that could be in the set
  - Bit is zero if corresponding value is not in set
  - Bit is one if corresponding value is in set
  - Intersect and union are just bit-level operations

- Worst case time complexity for sequence ADT implementations

	`add`	`remove`	`contains`	`intersection`	`union`
list	$O(N)$	$O(N)$	$O(N)$	$O(N \times M)$	$O(M+N)$
vector	$O(N)$	$O(N)$	$O(N)$	$O(N \times M)$	$O(M+N)$
lut	$O(1)$	$O(1)$	$O(1)$	$O(K)$	$O(K)$
bvec	$O(1)$	$O(1)$	$O(1)$	$O(K)$	$O(K)$

K = number of possible values in set

Set Uses

- Job scheduling
  - Use a set to represent resources required by a job
  - Can two jobs be executed at the same time? intersect
  - Combined resources require by two jobs? union
- Some algorithms need to track processed items in a data structure
  - Scan through sequence to find minimum element
  - Copy minimum element to output sequence
  - Use set to track which elements have been copied
  - Next scan skips over elements that are also in set

5. Map ADTs

- Imagine we want a contact list mapping friends to phone numbers
- We need to be able to add a new friend and their number
- We need to be able to remove a friend and their number
- We need to be able to see if list contains a friend/number pair
- We need to be able to use a friend's name to lookup a number
- We don't care about the order of entries in the contact list

Map Interface

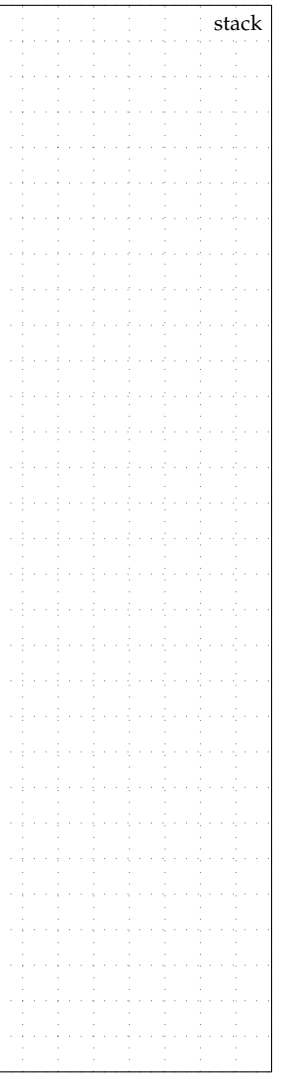
- Pseudocode for working with a map ADT

```
add    < "alice", 10 > to map
add    < "bob",   11 > to map
add    < "chris", 12 > to map
add    < "bob",   13 > to map
if map contain "bob" then
  set x to ( lookup "bob" in map )
```

- C-based interface for map ADT

```
typedef struct
{
  // opaque
}
map_t;

typedef /* user defined */ key_t;
typedef /* user defined */ value_t;

void    map_construct ( map_t* map );
void    map_destruct  ( map_t* map );
void    map_add       ( map_t* map, key_t k, value_t v );
void    map_remove    ( map_t* map, key_t k );
int     map_contains  ( map_t* map, key_t k );
value_t map_lookup    ( map_t* map, key_t k );
```

```
1  int main( void )
2  {
3    map_t map;
4    map_construct ( &map );
5    map_add       ( &map, "alice", 10 );
6    map_add       ( &map, "bob",   11 );
7    map_add       ( &map, "chris", 12 );
8    map_add       ( &map, "bob",   13 );
9
10   if ( map_contains( &map, "bob" ) ) {
11     int x = map_lookup( &map, "bob" );
12   }
13
14   map_destruct  ( &map );
15   return 0;
16 }
```



Map Implementation

- List implementation
  - Need new node type that can hold both key and value
  - `map_add` need to search list first for key ...
  - ... if key not in list then add to end of list (`list_push_back`)
  - `map_remove` needs to search list for *key*
  - `map_contains` needs to search list for *key*
  - `map_lookup` needs to search list for *key* return *value*
- Vector implementation
  - Need new `struct` type that can hold both key and value
  - Use an array of these `structs`
  - `map_add` need to search vector first for key ...
  - ... if key not in list then add to end of vector (`vector_push_back`)
  - `map_remove` needs to search vector for *key*
  - `map_contains` needs to search vector for *key*
  - `map_lookup` needs to search vector for *key* return *value*
- Lookup Table Implementation
  - Use a vector which is indexed by the key we want to store in map
  - Element is the corresponding value
  - Need way to indicate there is no value associated with a key
  - ... could use a set!
  - Element is one if the corresponding value is in map
  - Only possible if keys can be transformed into integers
  - Can be efficient if range of possible keys in set are small

- Worst case time complexity for sequence ADT implementations

	`add`	`remove`	`contains`	`lookup`
list	$O(N)$	$O(N)$	$O(N)$	$O(N)$
vector	$O(N)$	$O(N)$	$O(N)$	$O(N)$
lut	$O(1)$	$O(1)$	$O(1)$	$O(1)$

K = number of possible keys

Map Uses

- Tracking information about processes
  - Map Job IDs to usernames and other metadata
- Tracking information about flights
  - Map flight numbers to route, time, carrier
  - Map cities to list of departing flight numbers
  - Map carriers to flight numbers