ECE 2400 Computer Systems Programming Spring 2026

Topic 10: Abstract Data Types

School of Electrical and Computer Engineering
Cornell University

revision: 2026-03-11-11-54

- An abstract data type (ADT) is a high-level conceptual specification of an interface for a data type described using
  - informal sketch
  - standardized document ← *this course*
  - formal mathematical definition
  - programming language construct
- In this course, we will use a standardized documentation template to answer the following questions:
  - What is the name of the ADT?
  - What items are stored in the ADT?
  - How are these items related to each other?
  - What are the possible operations that can be performed on the ADT?
- A data structure is a concrete implementation of an ADT
- For each ADT we will:
  - sketch the high-level idea using an analogy
  - provide a standardized document describing the ADT
  - provide an example C-based interface for the ADT
  - discuss implementation trade-offs for the ADT

1. Indexed Sequence

Use Cases and/or Analogy:

- a streaming service's episode list for a particular show
- a photo slideshow with 1 photo on each page
- a text editor storing a document as sequence of characters

Indexed Sequence ADT

Collection of items ordered by `insert` operations

`insert(s,i,v)`	insert item `v` into sequence `s` at index `i`
`remove(s,i)`	remove item from sequence `s` at index `i`
`at(s,i)`	access item in sequence `s` at index `i`

1.1. Indexed Sequence Interface

```
1  typedef struct
2  {
3    // implementation defined
4  }
5  idxseq_t;
6
7  typedef /* any type */ item_t;
8
9  void    idxseq_construct ( idxseq_t* this );
10 void    idxseq_destruct  ( idxseq_t* this );
11 void    idxseq_insert    ( idxseq_t* this, int idx, item_t v );
12 void    idxseq_remove    ( idxseq_t* this, int idx );
13 item_t* idxseq_at        ( idxseq_t* this, int idx );
```

Example of using indexed sequence interface

```
idxseq_t idxseq;
idxseq_construct ( &idxseq );
idxseq_insert    ( &idxseq, 1, 12 );
idxseq_insert    ( &idxseq, 2, 18 );
idxseq_insert    ( &idxseq, 3, 21 );
idxseq_insert    ( &idxseq, 4, 45 );
...

for ( int i = 0; i < n; i++ )
  int v = *idxseq_at(i);

idxseq_destruct  ( &idxseq );
```

1.2. Indexed Sequence Implementation

- What if we implement the indexed sequence with either a doubly linked list or a resizable vector.
- What is the corresponding time complexity of these operations?

Indexed Sequence	**Time Complexity**	
	List	Vector
`insert`		
`remove`		
`at`		
`scan the sequence`		

2. Iterable Sequence

Use Cases and/or Analogy:

- making a music DVD with songs in a specific order
- dealing cards from a deck

Iterable Sequence ADT

Collection of items ordered by `insert` operations

`insert(s,i,v)`	insert item `v` into sequence `s` at iterator `i`
`remove(s,i)`	remove item from sequence `s` at iterator `i`
`begin(s)`	return iterator to beginning of sequence `s`
`end(s)`	return iterator to end of sequence `s`
`next(s,i)`	return iterator to next item in sequence `s`
`get(s,i)`	return item in sequence `s` at iterator `i`

2.1. Iterable Sequence Interface

```
1  typedef struct { /* implementation defined */ } itrseq_t;
2
3  typedef /* any type */                item_t;
4  typedef /* implementation defined */ itr_t;
5
6  void    itrseq_construct ( itrseq_t* this );
7  void    itrseq_destruct  ( itrseq_t* this );
8  void    itrseq_insert    ( itrseq_t* this, itr_t itr );
9  void    itrseq_remove    ( itrseq_t* this, itr_t itr );
10 itr_t   itrseq_begin     ( itrseq_t* this );
11 itr_t   itrseq_end       ( itrseq_t* this );
12 itr_t   itrseq_next      ( itrseq_t* this, itr_t itr );
13 item_t* itrseq_get       ( itrseq_t* this, itr_t itr );
```

Example of using iterable sequence interface

```
1  itrseq_t itrseq;
2  itrseq_construct ( &itrseq );
3  itrseq_insert    ( &itrseq, itrseq_end(&itrseq), 2 );
4  itrseq_insert    ( &itrseq, itrseq_end(&itrseq), 4 );
5  itrseq_insert    ( &itrseq, itrseq_end(&itrseq), 6 );
6  itrseq_insert    ( &itrseq, itrseq_end(&itrseq), 3 );
7  ...
8
9  itr_t itr = itrseq_begin( &itrseq );
10 while ( itr != itrseq_end( &itrseq ) ) {
11   int v = *itrseq_get( &itrseq, itr );
12   itr = itrseq_next( &itrseq, itr );
13 }
14
15 itrseq_destruct  ( &itrseq );
```

2.2. Iterable Sequence Implementation

- What should each of these operations return?

	List	Vector
`itr_t`	pointer to node	index
`begin`		
`end`		
`next`		
`get`		

- What is the corresponding time complexity of these operations?

Iterable Sequence	**Time Complexity**	
	List	Vector
`insert`		
`remove`		
`begin`		
`end`		
`next`		
`get`		
`scan the sequence`		

3. Stack

Use Cases and/or Analogy:

- pile of laundry?
- discard pile in a card game:
  - can add (push) cards onto the top of the pile
  - can remove (pop) cards from the top of the pile
  - cannot insert cards into the middle of the pile
  - only the top of the pile is accessible

Stack ADT

Collection of items in last in, first out (LIFO) order

`push(s,v)`	add item `v` to *top* of stack `s`
`pop(s)`	remove and return item from *top* of stack `s`

3.1. Stack Interface

```
1  typedef struct
2  {
3    // implementation defined
4  }
5  stack_t;
6
7  typedef /* any type */ item_t;
8
9  void   stack_construct ( stack_t* this );
10 void   stack_destruct  ( stack_t* this );
11 void   stack_push      ( stack_t* this, item_t v );
12 item_t stack_pop       ( stack_t* this );
```

Example of using stack interface

```
stack_t stack;
stack_construct   ( &stack );
stack_push        ( &stack, 6 );
stack_push        ( &stack, 2 );  // stack now has 2 items

int a = stack_pop ( &stack );     // returns 2
stack_push        ( &stack, 8 );
stack_push        ( &stack, 3 );  // stack now has 3 items

int b = stack_pop ( &stack );     // returns 3
int c = stack_pop ( &stack );     // returns 8
int d = stack_pop ( &stack );     // returns 6

stack_destruct    ( &stack );
```

3.2. Stack Implementation

- How can we implement the stack operations using either a doubly linked list or a resizable vector?
- What is the corresponding time complexity of these operations?

	Time Complexity	
Stack	List	Vector
`push`		
`pop`		

4. Queue

Use Cases and/or Analogy:

- calling customer service and being put on hold ("you are number 7")
- a queue of people waiting for coffee at CTB
  - 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

Queue ADT

Collection of items in first in, first out (FIFO) order

`enq(q,v)`	enqueue an item `v` to *tail* of queue `q`
`deq(q)`	dequeue and return item from *head* of queue `q`

4.1. Queue Interface

```
1  typedef struct
2  {
3    // implementation defined
4  }
5  queue_t;
6
7  typedef /* any type */ item_t;
8
9  void   queue_construct ( queue_t* this );
10 void   queue_destruct  ( queue_t* this );
11 void   queue_enq       ( queue_t* this, item_t v );
12 item_t queue_deq       ( queue_t* this );
```

Example of using queue interface

```
1  queue_t queue;
2  queue_construct   ( &queue );
3  queue_enq         ( &queue, 6 );
4  queue_enq         ( &queue, 2 );  // queue now has 2 items
5
6  int a = queue_deq ( &queue );     // returns 6
7  queue_enq         ( &queue, 8 );
8  queue_enq         ( &queue, 3 );  // queue now has 3 items
9
10 int b = queue_deq ( &queue );     // returns 2
11 int c = queue_deq ( &queue );     // returns 8
12 int d = queue_deq ( &queue );     // returns 3
13
14 queue_destruct    ( &queue );
```

4.2. Queue Implementation

- How can we implement the queue operations using either a doubly linked list or a resizable vector?
- What is the corresponding time complexity of these operations?

	Time Complexity	
Queue	List	Vector
`enq`		
`deq`		

5. Priority Queue

Use Cases and/or Analogy:

- boarding a plane (military, 1st class, families w/young children, etc.)
- Managing an emergency room at a hospital
  - Patients arrive and the triage nurse assigns each patient a priority
  - The triage nurse inserts patients into the waitlist based on priority
  - The emergency room doctor extracts patients from the waitlist based on priority; highest priority is always seen first

Priority Queue ADT

Collection of ⟨item,priority⟩ pairs

`insert(q,v,p)`	insert an item `v` with priority `p` into priority queue `q`
`extract(q)`	extract and return item with the highest priority from priority queue `q`

5.1. Priority Queue Interface

```
1  typedef struct
2  {
3    // implementation defined
4  }
5  pqueue_t;
6
7  typedef /* any type        */ item_t;
8  typedef /* comparable type */ pri_t;
9
10 void   pqueue_construct ( pqueue_t* this );
11 void   pqueue_destruct  ( pqueue_t* this );
12 void   pqueue_insert    ( pqueue_t* this, item_t v, pri_t p );
13 item_t pqueue_extract   ( pqueue_t* this );
```

Example of using priority queue interface

```
pqueue_t pqueue;
pqueue_construct          ( &pqueue );

pqueue_insert             ( &pqueue, "bob",   5 );
pqueue_insert             ( &pqueue, "cara",  7 );
pqueue_insert             ( &pqueue, "alice", 1 );

char* a = pqueue_extract ( &pqueue ); // returns "alice"
char* b = pqueue_extract ( &pqueue ); // returns "bob"
char* c = pqueue_extract ( &pqueue ); // returns "cara"

pqueue_destruct           ( &pqueue );
```

5.2. Priority Queue Implementation

- How can we implement the priority queue operations using either a doubly linked list or a resizable vector?
- What is the corresponding time complexity of these operations?
- What if we keep list or vector sorted by priority?

	Time Complexity			
Priority Queue	List (unsorted)	Vector (unsorted)	List (sorted)	Vector (sorted)
`insert`				
`extract`				

6. Set

Use Cases and/or Analogy:

- Shopping at Greenstar with your roommate for shared items
  - each of you has your own shopping bag
  - add items to your shopping bag
  - remove items from your shopping bag
  - might need to see if your bag already contains an item
  - might want to see if you both grabbed the same item (intersect)
  - might want to combine bags before checkout (union)

Set ADT

Collection of items

`add(s,v)`	add item `v` to set `s` if not already in set
`remove(s,v)`	remove item `v` from set `s`
`contains(s,v)`	return true if item `v` is in set `s`
`intersect(s,s0,s1)`	make set `s` intersection of sets `s0` and `s1`
`union(s,s0,s1)`	make set `s` union of sets `s0` and `s1`

6.1. Set Interface

```
1  typedef struct { /* implementation defined */ } set_t;
2  typedef /* any type */ item_t;
3
4  void set_construct ( set_t* this );
5  void set_destruct  ( set_t* this );
6  void set_add       ( set_t* this, item_t v );
7  void set_remove    ( set_t* this, item_t v );
8  int  set_contains  ( set_t* this, item_t v );
9  void set_intersect ( set_t* this, set_t* s0, set_t* s1 );
10 void set_union     ( set_t* this, set_t* s0, set_t* s1 );
```

Example of using set interface

```
set_t set;
set_construct ( &set );
set_add       ( &set, 2 );
set_add       ( &set, 4 );
set_add       ( &set, 6 );

int x = set_contains( &set, 4 );

set_destruct  ( &set );
```

6.2. Set Implementation

- How can we implement the set operations using either a doubly linked list or a resizable vector?
- What is the corresponding time complexity of these operations?
- What if we maintain a sorted doubly linked list or resizable vector?

	Time Complexity			
Set	List (unsorted)	Vector (unsorted)	List (sorted)	Vector (sorted)
`add`				
`remove`				
`contains`				

7. Map

Use Cases and/or Analogy:

- store database that maps a barcode to product details and pricing
- contact list mapping friends to phone numbers
  - need to add a new friend and their number
  - need to remove a friend and their number
  - need to use a friend's name to lookup a number
  - don't care about the order of entries in the contact list

Map ADT

Collection of ⟨key,item⟩ pairs

`add(m,k,v)`	if key `k` is already in map `m`, update item to `v`, otherwise add ⟨`k`,`v`⟩ pair to map `m`
`remove(m,k)`	remove ⟨`k`,`v`⟩ pair from map `m`
`lookup(m,k)`	find ⟨`k`,`v`⟩ pair in map `m` and return item `v`

7.1. Map Interface

```
1  typedef struct { /* implementation defined */ } map_t;
2
3  typedef /* any type */ key_t;
4  typedef /* any type */ item_t;
5
6  void   map_construct ( map_t* this );
7  void   map_destruct  ( map_t* this );
8  void   map_add       ( map_t* this, key_t k, item_t v );
9  void   map_remove    ( map_t* this, key_t k );
10 item_t map_lookup    ( map_t* this, key_t k );
```

Example of using map interface

```
map_t map;
map_construct ( &map );
map_add       ( &map, "alice", 10 );
map_add       ( &map, "bob",   11 );
map_add       ( &map, "cara",  12 );
map_add       ( &map, "bob",   13 );

int x = map_lookup( &map, "bob" ) )

map_destruct  ( &map );
```

7.2. Map Implementation

- How can we implement the map operations using either a doubly linked list or a resizable vector?
- What is the corresponding time complexity of these operations?
- What if we maintain a sorted doubly linked list or resizable vector?

	Time Complexity			
Set	List (unsorted)	Vector (unsorted)	List (sorted)	Vector (sorted)
`add`				
`remove`				
`lookup`				

8. ADT Summary

ADT	Collection of ...	Operations
Indexed Sequence	items ord. by `insert` ops	`insert`, `remove`, `at`
Iterable Sequence	items ord. by `insert` ops	`insert`, `remove` `begin`, `end`, `next`, `get`
Stack	items in LIFO order	`push`, `pop`
Queue	items in FIFO order	`enq`, `deq`
Priority Queue	⟨item,priority⟩ pairs	`insert`, `extract`
Set	items	`add`, `remove`, `contains`, `union`, `intersect`
Map	⟨key,item⟩ pairs	`add`, `remove`, `lookup`

	Implementation					
ADT	List	Vector	Binary Search Tree	Binary Heap Tree	Lookup Table	Hash Table
Indexed Seq	✓	★				
Iterable Seq	★	★				
Stack	★	★				
Queue	★	★				
Priority Queue	✓	✓		★		
Set	✓	✓	★		★	★
Map	✓	✓	★		★	★

Trees and Tables can also be used on their own as ADTs
Graphs are a new ADT with specialized implementations