# Topic 9: Abstract Data Types

School of Electrical and Computer Engineering  
Cornell University

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequence ADTs</td>
</tr>
<tr>
<td>2</td>
<td>Stack ADTs</td>
</tr>
<tr>
<td>3</td>
<td>Queue ADTs</td>
</tr>
<tr>
<td>4</td>
<td>Set ADTs</td>
</tr>
<tr>
<td>5</td>
<td>Map ADTs</td>
</tr>
</tbody>
</table>
1. Sequence ADTs

- 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

1. insert 2 at beginning of sequence
2. insert 4 at end of sequence
3. insert 6 at end of sequence
4. insert 3 at beginning of sequence
5. set iterator to beginning of sequence
6. while iterator is not equal to end of sequence
   7. get value at iterator
   8. set iterator to next iterator

• C-based interface for sequence ADT

```c
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. Sequence ADTs

```c
int main( void )
{
    seq_t seq;
    seq_construct ( &seq );
    seq_insert ( &seq, seq_begin(&seq), 2 );
    seq_insert ( &seq, seq_end (&seq), 4 );
    seq_insert ( &seq, seq_end (&seq), 6 );
    seq_insert ( &seq, seq_begin(&seq), 3 );

    itr_t itr = seq_begin( &seq );
    while ( itr != seq_end( &seq ) )
    {
        int value = seq_get( &seq, itr );
        itr = seq_next( &seq, itr );
    }

    seq_destruct ( &seq );
    return 0;
}
```

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++`
2. Stack ADTs

- Worst case time complexity for sequence ADT implementations

<table>
<thead>
<tr>
<th></th>
<th>begin</th>
<th>end</th>
<th>next</th>
<th>insert</th>
<th>remove</th>
<th>get</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>vector</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

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

```plaintext
1. push 6 onto stack
2. push 2 onto stack
3. pop from stack
4. push 8 onto stack
5. push 3 onto stack
6. pop from stack
7. pop from stack
8. pop from stack
```

• C-based interface for stack ADT

```c
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 );
```
```c
int main( void )
{
    stack_t stack;
    stack_construct ( &stack );
    stack_push ( &stack, 6 );
    stack_push ( &stack, 2 );
    // Stack now has two items

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

    int b = stack_pop ( &stack );
    int c = stack_pop ( &stack );
    int d = stack_pop ( &stack );
    // Stack is now empty

    stack_destruct ( &stack );
    return 0;
}
```
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

<table>
<thead>
<tr>
<th></th>
<th>push</th>
<th>pop</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>vector</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

(*) amortized
2. Stack ADTs

Stack Uses

• Parsing HTML document, need to track currently open tags

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

```
1 enq 6 onto tail of queue
2 enq 2 onto tail of queue
3 deq from head of queue
4 enq 8 onto tail of queue
5 enq 3 onto tail of queue
6 deq from head of queue
7 deq from head of queue
8 deq from head of queue
```
3. Queue ADTs

- C-based interface for queue ADT

```c
typedef struct
{
    // opaque
}
queue_t;

typedef /* user defined */ item_t;

void queue_construct ( queue_t* queue_p );
void queue_destruct ( queue_t* queue_p );
void queue_enq       ( queue_t* queue_p, item_t v );
item_t queue_deq     ( queue_t* queue_p );
int    queue_empty   ( queue_t* queue_p );
```
3. Queue ADTs

```c
int main( void )
{
    queue_t queue;
    queue_construct( &queue );
    queue_enq( &queue, 6 );
    queue_enq( &queue, 2 );
    // Queue now has two items

    int a = queue_deq( &queue );
    queue_enq( &queue, 8 );
    queue_enq( &queue, 3 );
    // Queue now has three items

    int b = queue_deq( &queue );
    int c = queue_deq( &queue );
    int d = queue_deq( &queue );
    // Queue is now empty

    queue_destruct( &queue );
    return 0;
}
```
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

<table>
<thead>
<tr>
<th></th>
<th>enq</th>
<th>deq</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>vector</td>
<td>$O(1)^*$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>circular buffer</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

(*) 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

1. add 2 to set0
2. add 4 to set0
3. add 6 to set0
4. does set0 contain 4?
5. add 6 to set1
6. add 5 to set1
7. set set2 to union of set0 and set1

• C-based interface for set ADT

```c
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 );
```
```c
int main( void )
{
    set_t set0;
    set_construct ( &set0 );
    set_add ( &set0, 2 );
    set_add ( &set0, 4 );
    set_add ( &set0, 6 );

    if ( set_contains( &set0, 4 ) ) {
        ... more code ...
    }

    set_t set1;
    set_construct ( &set1 );
    set_add ( &set1, 4 );
    set_add ( &set1, 6 );

    set_t set3;
    set_union ( &set3, &set0, &set1 );

    set_destruct ( &set0 );
    set_destruct ( &set1 );
    set_destruct ( &set2 );
    return 0;
}
```
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
4. Set ADTs

- **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**

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>remove</th>
<th>contains</th>
<th>intersection</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N \times M)$</td>
<td>$O(M + N)$</td>
</tr>
<tr>
<td>vector</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N \times M)$</td>
<td>$O(M + N)$</td>
</tr>
<tr>
<td>lut</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(K)$</td>
<td>$O(K)$</td>
</tr>
<tr>
<td>bvec</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(K)$</td>
<td>$O(K)$</td>
</tr>
</tbody>
</table>

$K = \text{number of possible values in set}$

**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

```c
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

```c
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 );
```
int main( void )
{
  map_t map;
  map_construct( &map );
  map_add( &map, "alice", 10 );
  map_add( &map, "bob", 11 );
  map_add( &map, "chris", 12 );
  map_add( &map, "bob", 13 );

  if ( map_contains( &map, "bob" ) ) {
    int x = map_lookup( &map, "bob" );
  }

  map_destruct( &map );
  return 0;
}
5. Map ADTs

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
5. Map ADTs

- Worst case time complexity for sequence ADT implementations

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>remove</th>
<th>contains</th>
<th>lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>vector</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>lut</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

$K = \text{number of possible keys}$

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