ECE 2400 Computer Systems Programming
Fall 2021

Topic 10: Abstract Data Types

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-29-21-37

1 Indexed Sequence ADT 4
   1.1. Indexed Sequence Interface ........................................ 4
   1.2. Indexed Sequence Implementation ................................ 5

2 Iterable Sequence ADT 6
   2.1. Iterable Sequence Interface ....................................... 6
   2.2. Iterable Sequence Implementation ............................... 7

3 Stack ADT 8
   3.1. Stack Interface ...................................................... 8
   3.2. Stack Implementation .............................................. 9
   3.3. Stack Applications .................................................. 9

4 Queue ADT 10
   4.1. Queue Interface ..................................................... 10
   4.2. Queue Implementation ............................................. 11
   4.3. Queue Applications ................................................ 12
zyBooks  The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

Copyright © 2021 Christopher Batten.  All rights reserved.  This handout was prepared by Prof.  Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming.  Download and use of this handout is permitted for individual educational non-commercial purposes only.  Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
• An **abstract data type** (ADT) is a high-level conceptual specification of an interface for a data type
  – an informal sketch
  – formal mathematical definition
  – programming language construct

• A **data structure** is a concrete implementation of an ADT

• In this topic, we will discuss seven ADTs:
  – Indexed Sequence    insert, remove, at
  – Iterable Sequence   insert, remove, begin, end, next, get
  – Stack               push, pop
  – Queue               enq, deq
  – Priority Queue      insert, extract
  – Set                 add, remove, contains, union, intersect
  – Map                 add, remove, lookup

• For each ADT we will:
  – **sketch** the high-level idea using an analogy
  – provide an example C-based **interface** for the ADT
  – discuss **implementation** trade-offs for the ADT
  – discuss **applications** for the ADT
1. Indexed Sequence ADT

- 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 songs at a position in the playlist

1.1. Indexed Sequence Interface

```c
typedef struct { /* implementation defined */ } idxseq_t;
typedef /* any type */ item_t;

void idxseq_construct ( idxseq_t* this );
void idxseq_destruct ( idxseq_t* this );
void idxseq_insert ( idxseq_t* this, int idx, item_t v );
void idxseq_remove ( idxseq_t* this, int idx );
item_t* idxseq_at ( idxseq_t* this, int idx );
```

Example of using indexed sequence interface

```c
idxseq_t idxseq;
idxseq_construct ( &idxseq );
idxseq_insert ( &idxseq, 1, 2 );
idxseq_insert ( &idxseq, 2, 4 );
idxseq_insert ( &idxseq, 3, 6 );
idxseq_insert ( &idxseq, 4, 3 );

for ( int i = 0; i < 4; i++ )
    int value = *idxseq_at(i);
idxseq_destruct ( &idxseq );
```
1.2. Indexed Sequence Implementation

- List implementation
  - All operations must step through each node in the list to reach the item with desired index (may need to step through entire list thus worst-case time complexity is $O(N)$)

- Vector implementation
  - `idxseq_insert`/`idxseq_remove` can directly index to desired element, but then must shift up/down remaining elements in vector (may need to shift all elements thus worst-case time complexity is $O(N)$)
  - `idxseq_at` can directly index to desired element (time complexity is $O(1)$)
2. Iterable Sequence ADT

• Same music playlist analogy now with a stronger emphasis on being able to iterate through the playlist to play the music

2.1. Iterable Sequence Interface

```c
typedef struct { /* implementation defined */ } itrseq_t;
typedef /* any type */ item_t;
typedef /* implementation defined */ itr_t;

void itrseq_construct ( itrseq_t* this );
void itrseq_destruct ( itrseq_t* this );
void itrseq_insert ( itrseq_t* this, itr_t itr );
void itrseq_remove ( itrseq_t* this, itr_t itr );
itr_t itrseq_begin ( itrseq_t* this );
itr_t itrseq_end ( itrseq_t* this );
itr_t itrseq_next ( itrseq_t* this, itr_t itr );
item_t* itrseq_get ( itrseq_t* this, itr_t itr );
```

Example of using iterable sequence interface

```c
itrseq_t itrseq;
itrseq_construct ( &itrseq );
itrseq_insert ( &itrseq, itrseq_end(&itrseq), 2 );
itrseq_insert ( &itrseq, itrseq_end(&itrseq), 4 );
itrseq_insert ( &itrseq, itrseq_end(&itrseq), 6 );
itrseq_insert ( &itrseq, itrseq_end(&itrseq), 3 );

itr_t itr = itrseq_begin( &itrseq );
while ( itr != itrseq_end( &itrseq ) ) {
    int value = *itrseq_get( &itrseq, itr );
    itr = itrseq_next( &itrseq, itr );
}

itrseq_destruct ( &itrseq );
```
2.2. Iterable Sequence Implementation

- List implementation
  - `itr_t` is a pointer to a node
  - `itrseq_begin` returns the head pointer
  - `itrseq_end` returns the NULL pointer
  - `itrseq_next` returns `itr->next_p`
  - `itrseq_get` returns `&(itr->value)`
  - Time complexity of all iterator operations is $O(1)$
  - `itrseq_insert`/`itrseq_remove` can directly manipulate pointers in doubly linked list thus time complexity is $O(1)$ regardless of location

- Vector implementation
  - `itr_t` is an index
  - `itrseq_begin` returns 0
  - `itrseq_end` returns size
  - `itrseq_next` returns `itr++`
  - `itrseq_get` returns `&(m_data[itr])`
  - Time complexity of all iterator operations is $O(1)$
  - `itrseq_insert`/`itrseq_remove` must shift up/down remaining elements in vector (may need to shift all elements thus worst-case time is $O(N)$)
3. Stack ADT

- 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)

3.1. Stack Interface

```c
typedef struct { /* implementation defined */ } stack_t;
typedef /* any type */ item_t;

void stack_construct ( stack_t* this );
void stack_destruct ( stack_t* this );
void stack_push ( stack_t* this, item_t v );
item_t stack_pop ( stack_t* this );
```

Example of using stack interface

```c
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

- List implementation
  - `stack_push` operates on back of list with `list_push_back`
  - `stack_pop` also operates on back of list with `list_pop_back`
  - Time complexity for both operations is $O(1)$
- Vector implementation
  - `stack_push` operates on back of vector with `vector_push_back`
  - `stack_pop` also operates on back of vector with `vector_pop_back`
  - Amortized time complexity for both operations is $O(1)$

3.3. Stack Applications

- 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> 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
4. Queue ADT

- Imagine a queue of people waiting for coffee at College Town Bagels
- People enqueue \texttt{(enq)} at the back of the line to wait
- People dequeue \texttt{(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)

4.1. Queue Interface

```c
typedef struct { /* implementation defined */ } queue_t;

typedef /* any type */ item_t;

void queue_construct ( queue_t* this );
void queue_destruct ( queue_t* this );
void queue_enq ( queue_t* this, item_t v );
item_t queue_deq ( queue_t* this );
```

Example of using queue interface

```c
queue_t queue;
queue_construct ( &queue );
queue_enq ( &queue, 6 );
queue_enq ( &queue, 2 ); // queue now has 2 items

int a = queue_deq ( &queue ); // returns 6
queue_enq ( &queue, 8 );
queue_enq ( &queue, 3 ); // queue now has 3 items

int b = queue_deq ( &queue ); // returns 2
int c = queue_deq ( &queue ); // returns 8
int d = queue_deq ( &queue ); // returns 3

queue_destruct ( &queue );
```
4.2. Queue Implementation

- **List implementation**
  - `queue_enq` operates on back of list with `list_push_back`
  - `queue_deq` operates on front of list with `list_pop_front`
  - Time complexity of both operations is $O(1)$

- **Vector implementation**
  - `queue_enq` operates on back of vector with `vector_push_back`
    (amortized time complexity is $O(1)$)
  - `queue_deq` operates on front of vector and always shifts down all elements with `vector_pop_front` (time complexity is $O(N)$)

- **Vector implementation as circular buffer**
  - 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
  - Amortized time complexity for both operations is $O(1)$
4.3. Queue Applications

• Network processing
  – Operating system provides queues for network interface 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
5. Priority Queue ADT

- Imagine we are 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

5.1. Priority Queue Interface

```c
typedef struct { /* implementation defined */ } pqueue_t;

typedef /* any type */ item_t;
typedef /* comparable type */ priority_t;

void pqueue_construct ( pqueue_t* this );
void pqueue_destruct ( pqueue_t* this );
void pqueue_insert ( pqueue_t* this, item_t v, priority_t p );
item_t pqueue_extract ( pqueue_t* this );
```

Example of using priority queue interface

```c
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 );
```

Topic 10: Abstract Data Types
5.2. **Priority Queue Implementation**

- **List implementation**
  - `pqueue_insert` scans list and inserts item to maintain sorted priority order with highest priority item at front (may need to scan entire list thus worst-case time complexity is $O(N)$)
  - `pqueue_extract` operates on the front of list with `list_pop_front` (time complexity is $O(1)$)

- **Vector implementation**
  - `pqueue_insert` adds item to back of vector with `vector_push_back` (amortized time complexity is $O(1)$)
  - `pqueue_extract` scans vector to find minimum priority item, then removes that time and shifts remaining items down (may need to scan entire vector thus worst-case time complexity is $O(N)$)

5.3. **Priority Queue Applications**

- **Job scheduling**
  - User gives each job a priority
  - Operating system places jobs in priority queue
  - Operating system schedules jobs on the machine based on priority

- **Discrete-event simulation**
  - Events are given a timestamp that they should occur in the future
  - Simulator places events into a priority queue
  - Simulator always chooses highest priority event (i.e., event that is supposed to happen next in time) to execute
  - Each event might generate more events that go into priority queue

- **Graph algorithms**
  - Dijkstra’s shortest path algorithm uses a priority queue
  - Prim’s minimum spanning tree algorithm uses a priority queue
6. Set ADT

- Imagine we are shopping at Greenstar with a friend
- Both of us 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 (intersect)
- We might want to combine our bags before we checkout (union)
- We don’t care about the order of items in the bag

6.1. Set Interface

```c
typedef struct { /* implementation defined */ } set_t;
typedef /* any type */ item_t;

void set_construct ( set_t* this );
void set_destruct ( set_t* this );
void set_add ( set_t* this, item_t v );
void set_remove ( set_t* this, item_t v );
int set_contains ( set_t* this, item_t v );
void set_intersect ( set_t* this, set_t* s0, set_t* s1 );
void set_union ( set_t* this, set_t* s0, set_t* s1 );
```

Example of using set interface

```c
set_t set0;
set_construct ( &set0 );
set_add ( &set0, 2 );
set_add ( &set0, 4 );
set_add ( &set0, 6 );

if ( set_contains ( &set0, 4 ) ) ...
```
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 );

6.2. Set Implementation

• List implementation
  – set_add need to search list first ...
  – ... if not in list then add to back of list with list_push_back
  – set_remove/set_contains also need to search list
  – set_intersect 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 vector first ...
  – ... if not in vector then add to back of vector with vector_push_back
  – set_remove needs to search vector, shift elements over
  – set_contains needs to search vector
  – set_intersect for each element in one vector, search other vector
  – set_union needs to iterate over both input vectors

• Time complexity
  – set_add, set_remove, set_contains may need to search the entire data structure and thus worst-case time complexity is $O(N)$
  – set_intersect is $O(N \times M)$
  – set_union is $O(N \times M)$ to avoid duplicates
6.3. Set Applications

- 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
# 7. Map ADT

- 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

## 7.1. Map Interface

```c
typedef struct { /* implementation defined */ } map_t;
typedef /* any type */ key_t;
typedef /* any type */ value_t;

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

### Example of using map interface

```c
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 );

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

map_destruct ( &map );
```
7.2. 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 back of list with `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 vector then add to back of vector with `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

- Time complexity
  - `map_add`, `map_remove`, `map_contains`, `map_lookup` all need to search the data structure and thus worst-case time complexity is $O(N)$

7.3. Map Applications

- 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
## 8. ADT Implementation Summary

### Implementation

<table>
<thead>
<tr>
<th>ADT</th>
<th>List</th>
<th>Vector</th>
<th>Binary Search Tree</th>
<th>Binary Heap Tree</th>
<th>Lookup Table</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed Seq</td>
<td>✓</td>
<td>✭</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterable Seq</td>
<td>✭</td>
<td>✭</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>✭</td>
<td>✭</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>✭</td>
<td>✭</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority Queue</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>★</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>✓</td>
<td>✓</td>
<td>★</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Map</td>
<td>✓</td>
<td>✓</td>
<td>★</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
</tbody>
</table>

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