ECE 2400 Computer Systems Programming
Fall 2019

Topic 10: Stacks, Queues, Sets, and Maps

School of Electrical and Computer Engineering
Cornell University

revision: 2019-10-18-09-06

1 Stacks 3
  1.1. Stack Interface .......................................................... 4
  1.2. Stack Implementation .................................................. 5
  1.3. Applications ............................................................. 6

2 Queues 7
  2.1. Queue Interface .......................................................... 8
  2.2. Queue Implementation .................................................. 9
  2.3. Applications ............................................................. 10

3 Sets 11
  3.1. Set Interface ............................................................. 12
  3.2. Set Implementation .................................................... 13
  3.3. Applications ............................................................. 14

4 Maps 15
  4.1. Map Interface ............................................................. 16
  4.2. Map Implementation .................................................... 17
Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.

Copyright © 2019 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.
1. Stacks

- In this topic, we will discuss four more data structures:
  - Stacks push, pop, empty
  - Queues enq, deq, empty
  - Sets add, remove, contains, union, intersect
  - Maps add, remove, lookup

- For each data structure we will:
  - sketch the high-level idea using an analogy and pseudocode
  - provide an example interface for the data structure
  - discuss implementation trade-offs for the data structure
  - discuss applications for the data structure

1. Stacks

- 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)

- Pseudocode for working with a stack

```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
```
1. Stacks

1.1. 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);
int stack_empty ( stack_t* this);
```

Example of using stack interface

```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;
}
```
1.2. Stack Implementation

- List implementation
  - `stack_push`/`stack_pop` operate on tail of list (`list_push_back`)
  - `stack_pop` should be straightforward to implement (`list_pop_back`)

- Vector implementation
  - `stack_push`/`stack_pop` operate on end of vector (`vector_push_back`)
  - `stack_pop` should be straightforward to implement (`vector_pop_back`)
  - What is worst case time complexity for `stack_push`?

- Worst case time complexity for stack 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
1.3. 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>
</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
2. Queues

- 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)

- Pseudocode for working with a queue

```
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
```
2.1. 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 );
int queue_empty ( queue_t* this );
```

Example of using queue interface

```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;
}
```
2.2. 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 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

2.3. Applications

• 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
3. Sets

- 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 (intersection)
- We might want to combine our bags before we checkout (union)
- We don’t care about the order of items in the bag

- Pseudocode for working with a set

```plaintext
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
```
3. Sets

3.1. 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
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 ) ) ... 

    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;
}
```
3. Sets

3.2. 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 vector
  - `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 set 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>

\(K = \text{maximum number of possible values in set}\)
3.3. 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 required 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
4. Maps

- 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

- Pseudocode for working with a map

```plaintext
1 add < "alice", 10 > to map
2 add < "bob", 11 > to map
3 add < "chris", 12 > to map
4 add < "bob", 13 > to map
5 if map contain "bob" then
6 set x to ( lookup "bob" in map )
```
4. Maps

4.1. 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
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;
}
```
4.2. 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 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 `struct`
  - `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
4. Maps

4.3. Applications

- Worst case time complexity for map 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>

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

4.3. 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