1 Sequence ADTs

1.1. Example C-Based Interface for Corresponding CDT

1.2. Trade-Offs in Choosing a CDT

2 Stack ADTs

2.1. Example C-Based Interface for Corresponding CDT

2.2. Trade-Offs in Choosing a CDT

2.3. Applications

3 Queue ADTs

3.1. Example C-Based Interface for Corresponding CDT

3.2. Trade-Offs in Choosing a CDT

3.3. Applications

4 Set ADTs

4.1. Example C-Based Interface for Corresponding CDT

4.2. Trade-Offs in Choosing a CDT

4.3. Applications
5 Map ADTs

5.1. Example C-Based Interface for Corresponding CDT ............... 23
5.2. Trade-Offs in Choosing a CDT ........................................... 25
5.3. Applications ................................................................. 26
• 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

• An ADT can usually be implemented using various different CDTs

• In this topic, we will discuss five 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

• For each ADT we will:
  – sketch the high-level idea using an analogy and pseudocode
  – provide an example C-based interface for a corresponding CDT
  – discuss trade-offs involved in choosing a CDT
  – discuss uses for the ADT
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

Pseudocode for working with a sequence ADT

```python
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
```
1.1. Example C-Based Interface for Corresponding CDT

```c
typedef struct
{
    // opaque
}

struct seq_t;

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

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

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;
}
```
1.2. **Trade-Offs in Choosing a CDT**

- **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**

<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><strong>list</strong></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><strong>vector</strong></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)

• Pseudocode for working with a stack ADT

```
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
```
2.1. Example C-Based Interface for Corresponding CDT

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

typedef /* user defined */ 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 );
```
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;
}
2.2. **Trade-Offs in Choosing a CDT**

- **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
2.3. 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>
</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)

• Pseudocode for working with a queue ADT

```plaintext
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
```
3.1. Example C-Based Interface for Corresponding CDT

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

typedef /* user defined */ 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 );
```
3. Queue ADTs

3.1. Example C-Based Interface for Corresponding CDT

```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;
}
```

stack
3.2. Trade-Offs in Choosing a CDT

- **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

3.3. 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
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
4. Set ADTs

4.1. Example C-Based Interface for Corresponding CDT

- Pseudocode for working with a set ADT

```c
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
typedef struct { /* opaque */ } set_t;
typedef /* user defined */ 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* set_src0, set_t* set_src1 );

void set_union ( set_t* this,
     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;
}
```
4.2. Trade-Offs in Choosing a CDT

- **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

4.3. Applications

- **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 = number of possible values in set

4.3. 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 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
5. Map ADTs

5.1. Example C-Based Interface for Corresponding CDT

- 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 )
```

5.1. Example C-Based Interface for Corresponding CDT

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

typedef /* user defined */ key_t;
typedef /* user defined */ 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 );
```
```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;
}
```
5.2. Trade-Offs in Choosing a CDT

- 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
5. Map ADTs

5.3. Applications

• 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>

$K =$ number of possible keys

5.3. 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