Abstract Data Types

Josh Wilcox (jw14g24@soton.ac.uk)

February 3, 2025

Contents

1 Encapsulation

1.1 In Object-Oriented Programming

- In object-oriented programming, encapsulation separates the interface from the implementation.
- The implementation is encapsulated and can be changed without affecting the usage of the class.
- The interface consists of the set of public methods.

1.2 In Abstract Data Types

- Abstract Data Types (ADTs) are similar to interfaces but are independent of programming languages.
- An ADT specifies the operations through which a data structure can be accessed.
- The purpose of an ADT is to allow for the declaration of intentions, abstracting away the implementation details.

2 Stacks

2.1 Description

- Last In First Out (LIFO) Memory
- Only allowed to add and remove items in the collection in a very limited way
- Can only see the very top item
- Can only add and remove from the top
- Implemented using an array or *linked list*

2.2 Standard Operations

- **Push** - Address an element to the top of the stack
- **Peek** - Allows to see the top element of the stack
- **Pop** - Removes the item from the top of the stack and returns it
- **IsEmpty** - Boolean determining whether the stack is empty

2.3 Array Implementation

- Making a stack with at most n elements using array `S[1..n]`
- `top` is the top index of the stack

2.3.1 Push

```
if (top < n) {
    top = top + 1
    S[top] = x
}
```

2.3.2 Pop

```
if (top==0){
    error
} else{
```

```
    top = top - 1
    return S[top+1] // Doing this after to allow the decrement
}
```

2.3.3 Peek

```
if (top==0){
    error
} else{
    return S[top]
}
```

2.3.4 Is-Empty

```
return top==0
```

2.4 Why use a Stack

- Provides a very simple interface
- Limits memory access to become non-random
- Sufficient for many applications
- Prevents programmers from using memory that may break existing code

2.5 Evaluating Arithmetic Expressions Using Stacks

For a previously fully-bracketed expression

1. Ignore any opening brackets
2. Repeat, moving left-to-right, until reaching the end:
    (a) Repeat until reaching a closing bracket:
        i. Add numbers to number stack
        ii. Add operators to operator stack
    (b) Pop the operator from the operator stack
    (c) Pop the required number of operands from the number stack
    (d) Apply the operator to the operands
    (e) Push the result back onto the number stack
3. The result will be the only number left in the number stack

2.6 Example

Consider the expression $(3 + (2 \times 5))$:

1. Ignore the opening bracket '('
2. Add 3 to the number stack
3. Add '+' to the operator stack
4. Ignore the opening bracket '('
5. Add 2 to the number stack

6. Add '*' to the operator stack
7. Add 5 to the number stack
8. Encounter closing bracket ')':
   (a) Pop '*' from the operator stack
   (b) Pop 5 and 2 from the number stack
   (c) Calculate $2 \times 5 = 10$
   (d) Push 10 onto the number stack
9. Encounter closing bracket ')':
   (a) Pop '+' from the operator stack
   (b) Pop 10 and 3 from the number stack
   (c) Calculate $3 + 10 = 13$
   (d) Push 13 onto the number stack
10. The result is 13

3 Queues

3.1 Description

- First-in-first-out (FIFO) memory model
- Acts like a queue of people, the first one entering the line will be the first to reach the end at all times

3.2 Standard Operations

3.2.1 Enqueue

- Adds an element to the end of the queue.

3.2.2 Peek

- Allows to see the front element of the queue without removing it.

3.2.3 Dequeue

- Removes the front element from the queue and returns it.

3.2.4 isEmpty

- Boolean to define if the queue is empty

3.3 Array Implementation

3.3.1 Enqueue

- Using an array `Q[1..n]` with `front` and `rear` indices.

```
if (rear < n) {
    rear = rear + 1;
    Q[rear] = x;
} else {
    error("Queue is full");
}
```

Big Theta: $\Theta(1)$ - Enqueue operation is constant time as it involves a simple increment and assignment.

3.3.2 Peek

- Returns the element at the `front` of the queue.

```
if (front == rear) {
    error("Queue is empty");
} else {
    return Q[front + 1];
}
```

Big Theta: $\Theta(1)$ - Peek operation is constant time as it involves accessing an element at a specific index.

3.3.3 Dequeue

- Removes and returns the element at the `front` of the queue.

```
if (front == rear) {
    error("Queue is empty");
} else {
    front = front + 1;
    return Q[front];
}
```

Big Theta: $\Theta(1)$ - Dequeue operation is constant time as it involves a simple increment and access.

3.3.4 isEmpty

- Checks if the queue is empty.

```
return front == rear;
```

Big Theta: $\Theta(1)$ - isEmpty operation is constant time as it involves a simple comparison.

4 Priority Queues

4.1 Description

- A priority queue is an abstract data type where each element has a priority.
- Elements with higher priority are dequeued before elements with lower priority.
- If two elements have the same priority, they are dequeued according to their order in the queue.

4.2 Standard Operations

4.2.1 Enqueue

- Adds an element to the queue with an associated priority.

4.2.2 Peek

- Allows to see the element with the highest priority without removing it.

4.2.3 Dequeue

- Removes and returns the element with the highest priority.

4.2.4 isEmpty

- Boolean to define if the priority queue is empty.

4.3 Implementation

- Implemented most efficiently using a **heap**
- Can also be implemented using a linked list or a binary tree

5 Lists

5.1 Description

- An ordered collection of elements.
- Elements can be accessed by their position (index) in the list.

5.2 Standard Operations

5.2.1 Add

- Adds an element to the end of the list.

5.2.2 Get

- Retrieves the element at a specified index.

5.2.3 Remove

- Removes the element at a specified index.

5.2.4 isEmpty

- Boolean to define if the list is empty.

5.3 Array Implementation

5.3.1 Add

```
if (size < n) {
    list[size] = x;
    size = size + 1;
} else {
    error("List is full");
}
```

5.3.2 Get

```
if (index >= 0 && index < size) {
    return list[index];
} else {
    error("Index out of bounds");
}
```

5.3.3 Remove

```
if (index >= 0 && index < size) {
    for (int i = index; i < size - 1; i++) {
        list[i] = list[i + 1];
    }
    size = size - 1;
} else {
    error("Index out of bounds");
}
```

5.3.4 isEmpty

```
return size == 0;
```

6 Sets

6.1 Description

- An unordered collection of unique elements.
- Does not allow duplicate elements.

6.2 Standard Operations

6.2.1 Add

- Adds an element to the set if it is not already present.

6.2.2 Contains

- Checks if the set contains a specified element.

6.2.3 Remove

- Removes a specified element from the set.

6.2.4 isEmpty

- Boolean to define if the set is empty.

6.3 Array Implementation

6.3.1 Add

```
if (!contains(set, x)) {
    if (size < n) {
        set[size] = x;
        size = size + 1;
    } else {
        error("Set is full");
    }
}
```

6.3.2 Contains

```
for (int i = 0; i < size; i++) {
    if (set[i] == x) {
        return true;
    }
}
return false;
```

6.3.3 Remove

```
for (int i = 0; i < size; i++) {
    if (set[i] == x) {
        for (int j = i; j < size - 1; j++) {
            set[j] = set[j + 1];
        }
        size = size - 1;
        return;
    }
}
error("Element not found");
```

6.3.4 isEmpty

```
return size == 0;
```

7 Maps

7.1 Description

- A collection of key-value pairs.
- Each key is unique and maps to a specific value.

7.2 Standard Operations

7.2.1 Put

- Adds a key-value pair to the map.

7.2.2 Get

- Retrieves the value associated with a specified key.

7.2.3 Remove

- Removes the key-value pair associated with a specified key.

7.2.4 ContainsKey

- Checks if the map contains a specified key.

7.2.5 isEmpty

- Boolean to define if the map is empty.

7.3 Array Implementation

7.3.1 Put

```
for (int i = 0; i < size; i++) {
    if (map[i].key == key) {
        map[i].value = value;
        return;
    }
}
if (size < n) {
    map[size].key = key;
    map[size].value = value;
    size = size + 1;
} else {
    error("Map is full");
}
```

7.3.2 Get

```
for (int i = 0; i < size; i++) {
    if (map[i].key == key) {
        return map[i].value;
    }
}
error("Key not found");
```

7.3.3 Remove

```
for (int i = 0; i < size; i++) {
    if (map[i].key == key) {
        for (int j = i; j < size - 1; j++) {
            map[j] = map[j + 1];
        }
        size = size - 1;
        return;
    }
}
error("Key not found");
```

7.3.4 ContainsKey

```
for (int i = 0; i < size; i++) {
    if (map[i].key == key) {
        return true;
    }
}
return false;
```

7.3.5 isEmpty

```
return size == 0;
```