ADT Journal

Keith Bittner

Eastern Oregon University

Table of Contents

**Introduction3**

**Arrays Versus Linked Lists3**

Advantages and Disadvantages of Arrays4

Advantages and Disadvantages of Linked Lists4

**Stacks4**

Definition5

Operations5

Uses of Stacks6

Implementations6

**Queues7**

Definition7

Operations8

Uses of Queues8

Implementations9

**Unordered Lists10**

Definition11

Operations11

Uses of Unordered Lists12

Implementations13

**Order Lists14**

Definition14

Operations14

Uses of Ordered Lists16

Implementations16

**Binary Search Trees18**

Definition18

Operations18

Uses of Binary Search Trees19

**References21**

ADT Journal

In Java, just like any computer programming language, the amount of details involved in any software system is extensive enough that for one individual to fully comprehend is fundamentally impossible. To simplify these details, software engineers use a concept called abstraction. Abstraction is the process of hiding implementation details of a program from a user while providing all the functionality associated with the program. An example of an abstraction is an interface.

Interfaces are used to hide the details of a data type also known as objects by making them abstract. While a data type has defined values and specific operations associated with them, abstract data types do not. Collections are a good example of an abstract data type. A collection is an object that collects and organizes other objects. Because there are no specific values and operations defined, a collection can be anything such as a collection of books, or a collection of integers.

Collections are implemented using a data structure. Simply put, a data structure is the instructions on how to use a collection such as arrays and linked lists. Interfaces are the blueprints for the data structure that implements a collection. Therefore, a programmer connects a collection to an interface and the interface to a data structure.

**Arrays Versus Linked Lists**

With any computer language, there are always advantages and disadvantages with choosing one data structure over another. This is even more apparent when choosing how to implement a chosen data structure. The question becomes does a programmer choose to use arrays or does the programmer chooses to use linked lists.

**Advantages and Disadvantages of Arrays**

One of the main advantages of using arrays is that a user can access any element contain in the array by referencing its index. Arrays can be used to represent multiple objects of the same type through the declaration of one variable. This means arrays are easy to manipulate and can store large data sets. Additionally, they can be used to implement other data structures such as queues and lists. However, since arrays are fixed, the size cannot be increased or decreased once the array is declared. So, it is crucial to know the exact size that is needed, though this just isn’t always the case. That leads programmers into either over- or under-allocating memory for the array.

**Advantages and Disadvantages of Linked Lists**

Linked lists being dynamic is the main advantage of its utilization. This means that the size of the list can be increased or decreased while the program itself is being executed. There is no need to declare the size in advance. Linked lists use nodes to reference elements in the list, meaning that insertion and deletion of list elements is made easy through reassigning node pointers. Additionally, as with arrays, they can be used to implement abstract data types such as queues and lists. One downside to using linked lists is that the node pointers require extra memory space to store the location data. Furthermore, unlike arrays, to access a specific element contained within, the linked list must be traversed through the nodes.

**Stacks**

Imagine a stack of plates in a cafeteria at a school or hospital. When clean plates are brought out, they are placed on the top of the existing stack. When a customer needs a plate, they remove one off the top of the stack. If the bottom plate is needed or wanted, the customer would have to remove all the plates off the stack to reach the bottom plate. This is the basic principle of how stacks work.

**Definition**

A Stack is a linear collection that has its elements added and removed in what is known as last in first out (LIFO). This means that the last element to be inserted into the stack, must be the first element to be removed. Stacks are generally visualized as vertical collections of data.

**Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| push(T element) | Adds an element to the top of the stack. | void | O(1) |
| pop() | Removes an element from the top of the stack. | T element | O(1) |
| peek() | Examines the element at the top of the stack. | T element | O(1) |
| isEmpty() | Determines if the stack is empty. | boolean | O(1) |
| size() | Determines the number of elements on the stack. | integer | O(1) |

T denotes a generic type and element represents an object of that type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Additional Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| expandCapacity() | Create a new stack with the contents of the old stack and increased size. | void | O(1) |

|  |  |
| --- | --- |
| **Exceptions** | |
| **Operation** | **Exceptions Thrown** |
| pop() | EmptyCollectionException |
| peek() | EmptyCollectionException |

**Uses for Stacks**

There are various real-world applications for stacks. An example of an array implementation is postfix expressions. In infix expressions, the entire expression is evaluated with operations being performed according to the Order of Operations in arithmetic. However, in postfix, the expression is scanned from left to right. When the program encounters an operand, the value is pushed onto the top of the stack. When the program encounters an operation, the top two elements on the stack are popped off, the operation is performed on them, then the result is pushed back on top of the stack. When the program reaches the end of the expression, the value left on the stack is the final result.

An example of an application that uses linked lists to implement a stack is a text editor. As data is being typed into the text editor, the data is saved as the data itself and the location of memory which its stored. This is done so functions like undo and redo can remove mistakes or put back information that was removed by accident. This increases the functionality of text editors.

**Implementations**

Stacks can be implemented using either arrays or linked lists. The best choice for implementation depends on the need for the data structure.

**Array implementation.** The array implementation uses all the standard operations that come with stacks with the addition of expandCapacity(). This operation is called upon when the user attempts to push an element onto the stack, but the stack is full. ExpandCapacity() will create a new stack larger than the original and copy the contents from the original into the new. The operation will then rename the new stack with the old stacks name. Additionally, in the array implementation, the entire array contents and the top of the stack must be kept track of through local variables inside the program.

**Linked list implementation.** The linked list implementation uses all the standard operations that come with stacks. Unlike arrays however, linked lists use nodes to store the element and memory address so elements can be added or removed from the front of the stack. When the stack is full, the expandCapacity() operation does not need to be executed, however extra care must be taken when using a linked list. This means the front of the linked list must be kept track of through a pointer (reference variable storing the memory location). When using pointers, an element added to the front of the stack must first point to the existing front node, then the old front node is pointed to the newly added element. If you execute this in opposite order, the user will lose all information stored within the stack. Additionally, when removing an element from the stack, all that needs to be executed is pointing the front of the stack to the next element in the stack. If pop() is executed on the stack prior to setting the front to the next element in the stack, the user will lose all information stored within the stack.

**Queues**

Picture a customer standing in line at the bank. He turns around to see more and more people getting in line behind him. The customer then looks to the front of the line and sees one person at a time leaving the line to be helped by a teller. This is how queues data work in programming.

**Definition**

A queue is a type of linear collection whose elements are added on one end and removed from the other end. This is commonly known as first in first out (FIFO) and while similar in ways to stacks, queues differ mostly in the fact they are visualized as horizontal collections.

**Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| enqueue(T element) | Adds an element to the rear of the queue. | void | O(1) |
| dequeue() | Removes an element from the front of the queue. | T element | O(1) |
| first() | Examines the element at the front of the queue. | T element | O(1) |
| isEmpty() | Determines if the queue is empty. | boolean | O(1) |
| size() | Determines the number of elements in the queue. | integer | O(1) |

T denotes a generic type and element represents an object of that type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Additional Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| expandCapacity() | Creates a new queue with the contents of the old queue and increased capacity. | void | O(n) |

|  |  |
| --- | --- |
| **Exceptions** | |
| **Operation** | **Exception Thrown** |
| dequeue() | EmptyCollectionException |
| first() | EmptyCollectionException |

**Uses for Queues**

Just as stacks, queues have their own real-world applications. An example of linked list implementation is a computer’s update utility. Updates are downloaded in a specified order, so the first update downloaded becomes the first update to be installed. If computer updates were installed using stacks, applications and software may not install properly or work properly. Another example of linked list implementation is a ticket counter. People are served at the counter from the order in which they entered the line.

An example of a circular array implementation is a customer service call center. A service representative may be able to receive four calls at the same time. The representative answers the first line while the other three wait their turn. Once they are done with call one and move on to call two, line one becomes open for another caller. However, the representative will only answer line one once they have finished answering the third and fourth lines. This process continues if there are calls coming in and lines open to enter the queue.

**Implementations**

Queues can be implemented just like stacks with arrays and linked lists. Array implementations of a queue can be fixed linear based or circular based. Fixed arrays implementations of queues are not nearly as efficient as when being implemented with a stack.

**Single array implementation.** This array implementation uses all the standard operations that come with queues. Since a single array implementation is fixed, the program must keep track of all the elements of the array, the front, and rear of the array. This is because after you perform a dequeue(), the entire contents of the queue must be shifted to the front of the array. If there is no room left because the rear element has reached the end of available space, enlarging the array is not considered a practical solution. Instead a different implementation of arrays should be utilized.

**Circular array implementation.** The circular array implementation uses all the standard operations that come with queues with the addition of expandCapacity(). This operation is called upon when the user attempts to enqueue an element onto the queue, but the queue is full. ExpandCapacity() will create a new queue larger than the original and copy the contents from the original into the new. The operation will then rename the new queue with the old queues name. Unlike single arrays, circular arrays must keep track of all the elements contained in the queue, the front, and the rear of the queue. By storing this data, the queue can operate in a revolving fashion. This means that aside from having to expand when needed, the front and rear of the queue will revolve through the array as elements are enqueued and dequeued while maintaining FIFO integrity.

**Linked list implementation.** The linked list implementation uses all the standard operations that come with queues. Unlike arrays however, linked lists use nodes to store the element and memory address so elements can be removed from the front of the queue and added to the rear. When the queue is full, the expandCapacity() operation does not need to be executed, however extra care must be taken when using a linked list. This means the front and rear of the linked list must be kept track of through pointers (reference variables storing the memory location). To add an element at the rear of a queue, the current rear must be stored in a temporary variable, the new node then must point to the previous node, finally rear is made to point to the new node. If these are executed in opposite order, it becomes possible to lose all the information contained within the queue. Additionally, when removing an element from the queue, all the needs to be executed is pointing the front of the queue to the next element. If dequeue() is executed on the queue prior to setting the front to the next element, the user will lose all information stored within the queue.

**Unordered Lists**

Look at a brand-new unopened deck of cards. If the deck is opened and the cards faces are viewed, by default they are in a certain order. If one was to shuffle the deck of cards, the deck becomes unordered according to their characteristics. However, the deck is still considered to be ordered but only by the way the person shuffling the deck chose. This is the basic principle behind unordered lists.

**Definition**

An unordered list is a linear collection that is versatile when compared to stacks and queues. Elements may be added and/or removed at the front, rear and middle of the list. The elements in an unordered list are kept in a particular order according to how the user determines the order of the elements.

**Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| removeFirst() | Remove the first element from the list. | T element | O(1) |
| removeLast() | Remove the last element from the list. | T element | O(n) |
| remove(T element) | Removes a particular element from the list. | T element | O(n) |
| first() | Examines the element at the front of the list. | T element | O(1) |
| last() | Examines the element at the rear of the list. | T element | O(1) |
| contains(T target) | Determines if the list contains a particular element. | boolean | O(n) |
| isEmpty() | Determines if the list is empty. | boolean | O(1) |
| size() | Determines the number of elements in the list. | integer | O(1) |

T denotes a generic type and element / target represent objects of that type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Additional Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| addToFront(T element) | Adds an element to the front of the list. | void | O(1) |
| addToRear(T element) | Adds an element to the rear of the list. | void | O(1) |
| addAfter(T element, T target) | Adds an element after a particular element already in the list. | void | O(n) |
| expandCapacity() | Create a new list with the contents of the old list and increased capacity. | void | O(n) |

T denotes a generic type and element / target represent objects of that type.

|  |  |
| --- | --- |
| **Exceptions** | |
| **Operation** | **Exception Thrown** |
| removeFirst() | EmptyCollectionException |
| removeLast() | EmptyCollectionException |
| remove(T element) | EmptyCollectionException  ElementNotFoundException |
| first() | EmptyCollectionException |
| last() | EmptyCollectionException |
| contains(T target) | EmptyCollectionException |

T denotes a generic type and element / target represent objects of that type.

**Uses for Unordered Lists**

A list of courses required for a student to graduate from college with a certain degree is an example of an array implementation. Being an array, there is a fixed number of courses that must be taken. However, the order of those course is set based on the colleges preference and not based off perhaps their course number.

A train yard and the trains contained within is a good example of a linked list implementation. A train represents the list of linked cars and the train cars represent individual nodes of the list. Train cars can be added in any order, at the front, middle, or rear of the train. The train yard turntable would act as a temporary node to aid in the adding or removing of cars. This would allow for the train to be reconnected without losing its structure.

**Implementations**

Unordered lists can be implemented just like stacks and queues with arrays or linked lists. Since arrays are fixed, they are not always the best way to implement a list. Generally, if the size of your list is going to be static, then array implementations work well. However, the purpose of lists is to be able to dynamically change the data contained within while the program is running. For these reasons, linked lists are the best way to implement unordered lists.

**Array implementation.** The array implementation uses all the standard operations that come with unordered lists with the addition of expandCapacity(). This operation is called upon when the user attempts to add an element anywhere into the list, but the list is full. ExpandCapacity() will create a new list larger than the original and copy the contents from the original into the new. The operation will then rename the new list with the old lists name. Additionally, in the array implementation, all the elements contained in the list, the front, and the rear must be kept track of. This is because after you perform removeFirst() or remove() the entire contents of the list must be shifted to the front of the array.

**Linked list implementation.** The linked list implementation uses all the standard operations that come with unordered lists. Unlike arrays however, linked lists use nodes to store the element and memory address so elements can be added and removed from the front, middle or rear of the list. When the list is full, the expandCapacity() operation does not need to be executed, however extra care must be taken when using linked lists. This means the front and rear of the linked list must be kept track of through pointers (reference variables storing the memory location). When adding an element to the front of the list, the node must first point to the existing front, then the front is pointed to the newly added element. If adding an element in the middle of the list, the node must first point to the next element, then the previous node points to the newly inserted element. To add an element at the rear of a list, the current rear must be stored in a temporary variable, the new node then must point to the previous node, finally rear is made to point to the new node. Additionally, when removing an element from the front of the list, all the needs to be executed is pointing the front of the list to the next element. If removing an element from the middle of the list, all that needs to be executed is pointing the previous node to the next node which removes the unwanted element. Furthermore, when removing an element from the rear of the list, the rear is set to point at the previous node. If any of these are executed in opposite order, it becomes possible to lose all the information contained within the list.

**Ordered Lists**

Consider the structure of a phone book. Names, phone numbers and address are stored based off the characteristic of alphabetizing the names. Ordered lists are structured this way according to some specific characteristic of the elements.

**Definition**

An ordered list is a linear collection that is quite like unordered lists except that the elements may be added to the list only where they belong in the order. The elements in an ordered list are kept in a particular order according to a specific characteristic such as alphabetically or numerical.

**Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| removeFirst() | Remove the first element from the list. | T element | O(1) |
| removeLast() | Remove the last element from the list. | T element | O(n) |
| remove(T element) | Removes a particular element from the list. | T element | O(n) |
| first() | Examines the element at the front of the list. | T element | O(1) |
| last() | Examines the element at the rear of the list. | T element | O(1) |
| contains(T target) | Determines if the list contains a particular element. | boolean | O(n) |
| isEmpty() | Determines if the list is empty. | boolean | O(1) |
| size() | Determines the number of elements in the list. | integer | O(1) |

T denotes a generic type and element / target represent objects of that type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Additional Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| addToFront(T element) | Adds an element to the front of the list. | void | O(1) |
| addToRear(T element) | Adds an element to the rear of the list. | void | O(1) |
| addAfter(T element, T target) | Adds an element after a particular element already in the list. | void | O(n) |
| expandCapacity() | Create a new list with the contents of the old list and increased capacity. | void | O(n) |

T denotes a generic type and element / target represent objects of that type.

|  |  |
| --- | --- |
| **Exceptions** | |
| **Operation** | **Exception Thrown** |
| removeFirst() | EmptyCollectionException |
| removeLast() | EmptyCollectionException |
| remove(T element) | EmptyCollectionException  ElementNotFoundException |
| first() | EmptyCollectionException |
| last() | EmptyCollectionException |
| contains(T target) | EmptyCollectionException |

T denotes a generic type and element / target represent objects of that type.

**Uses for Ordered Lists**

An individual’s web browsing history is an example of a linked list implementation. Websites are stored in the order they were viewed based on the date and time. This allows a user to view all the sites they have visited over a given time period. While the user can delete any visited website from the browser history, adding a new site to the history can only be after the previous date and time.

A dictionary is a prime example of an array implementation of an ordered list. Woords are ordered alphabetically which allows an individual to be able to search and find a particular word contained within. Additions to the dictionary can only happen at the exact place it should be based on the first and subsequently letters in the word. You can not add words to just anywhere in the dictionary. An example of this would be trying to place the word cat after the word dog. If a individual needed to know to spell or the definition of cat, they would not find the word if it was just added anywhere in the dictionary.

**Implementations**

Ordered lists can be implemented just like unordered lists with arrays and linked lists. Since arrays are fixed, they are not always the best way to implement a list. Generally, if the size of your list is going to be static, then array implementations work well. However, the purpose of lists is to be able to dynamically change the data contained within while the program is running. For these linked lists are the best way to implement ordered lists.

**Array implementation.** The array implementation uses all the standard operations that come with ordered lists with the addition of expandCapacity(). This operation is called upon when the user attempts to add an element into the list, but the list is full. ExpandCapacity() will create a new list larger than the original and copy the contents from the original into the new. The operation will then rename the new list with the old lists name. Additionally, in the array implementation, all the elements contained in the list, the front, and the rear must be kept track of. This is because after you perform removeFirst() or remove() the entire contents of the list must be shifted to the front of the array.

**Linked list implementation.** The linked list implementation uses all the standard operations that come with ordered lists. Unlike arrays however, linked lists use nodes to store the element and memory address so elements can be added and removed from the list. When the list is full, the expandCapacity() operation does not need to be executed, however extra care must be taken when using linked lists. This means the front and rear of the linked list must be kept track of through pointers (reference variables storing the memory location). Elements in an ordered list are added where they specifically fit in according to the list characteristics. However, the operations are similar to unordered lists except all cases are contained within one function. When adding an element to the front of the list, the node must first point to the existing front, then the front is pointed to the newly added element. If adding an element in the middle of the list, the node must first point to the next element, then the previous node points to the newly inserted element. To add an element at the rear of a list, the current rear must be stored in a temporary variable, the new node then must point to the previous node, finally rear is made to point to the new node. Additionally, when removing an element from the front of the list, all the needs to be executed is pointing the front of the list to the next element. If removing an element from the middle of the list, all that needs to be executed is pointing the previous node to the next node which removes the unwanted element. Furthermore, when removing an element from the rear of the list, the rear is set to point at the previous node. If any of these are executed in opposite order, it becomes possible to lose all the information contained within the list.

**Binary Search Trees**

**Definition**

A search tree is a tree whose elements are organized to allow finding a element when needed. Elements in a binary search tree are stored so that the left nodes are elements that are less then it’s parent. The right nodes are elements that are greater than or equal to their parent.

**Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| getRootElement() | Examines the element at the root of the tree. | T element | O(1) |
| isEmpty() | Determines if the tree is empty. | boolean | O(1) |
| size() | Determines the number of elements in the tree. | integer | O(1) |
| contains(T target) | Determines if the tree contains a particular element. | boolean | O(n) |
| find() | Determines the location of a particular element in the tree. | T element | O(n) |

T denotes a generic type and element / target represent objects of that type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Additional Operations** | | | |
| **Name** | **Description** | **Return** | **Big ‘O’** |
| addElement(T element) | Adds an element to the tree. | void | O(n) |
| removeElement(T target) | Removes an element from the tree. | T element | O(n) |
| removeAllOccurences(T target) | Remove all occurrences of an element from the tree. | void | O(n) |
| removeMin() | Remove the minimum element in the tree. | T element | O(n) |
| removeMax() | Remove the maximum element in the tree. | T element | O(n) |
| findMin() | Determines the location of the minimum element in the tree. | T element | O(n) |
| findMax() | Determines the location of the maximum element in the tree. | T element | O(n) |

T denotes a generic type and element / target represent objects of that type.

|  |  |
| --- | --- |
| **Exceptions** | |
| **Operation** | **Exception Thrown** |
| getRootElement() | EmptyCollectionException |
| contains(T target) | ElementNotFoundException |
| find(T target) | ElementNotFoundException |
| removeElement() | ElementNotFoundException |
| removeAllOccurrences(T target) | ElementNotFoundException |
| removeMin() | EmptyCollectionException |
| removeMax() | EmptyCollectionException |
| findMin() | EmptyCollectionException |
| findMax() | EmptyCollectionException |

T denotes a generic type and element represents an object of that type.

**Uses for Binary Search Trees**

A great example of a binary search tree is Morse Code. The Morse Code alphabet a designed as a binary tree where the root node is an empty space and the first two children of the tree are the letters ‘E’ and ‘T’. Children nodes to the left of their parents are represented by their parents position and an additional dot. Children nodes to the right of their parents are represented by their parents position and an additional dash. For example, going down the entire left side of the Morse Code tree, you have the letters E, I, S, H, which are displayed as: \*; \*\*, \*\*\*, \*\*\*\*. According to this principle H has 3 dots which represent the letter S its parent plus an additional dot. By knowing that left nodes are dots and right nodes are dashes, an individual could locate any letter in the Morse Code tree and decipher any code.

References

Abstract Data Type. (2003). Encyclopedia of Computer Science.

Aho, A., & Ullman, J. D. (1992). *Foundations of computer science* (Principles of computer science series). New York: Computer Science Press.

Cornell University, (2006). *Abstraction*. Retrieved from Cornell University website: <https://www.cs.cornell..edu/courses/cs211/2006sp/Lectures/L08-Abstraction/08_abstraction.html>

Guttag, J. (1977). Abstract data types and the development of data structures. *Communications of the ACM, 20*(6), 396-404.

Jenkins, M. (1998). *Abstract data types in Java* (McGraw-Hill Java masters). New York: McGraw-Hill.

Lewis, J., DePasquale, P. J., & Chase, J. (2016). *Java Foundations: Introduction to Program Design & Data Structures*. Boston: Pearson.