Module 2: Array-based Structures

Overview

In this module, the processes of linked lists, recursion, stacks and queues will be tackled. You will learn the different processes of each linked list and the difference between stack and queue. You will also learn what is recursion in programming and where it can be applied.

Module Objectives

At the end of this module, the students are expected to:

1. Create a data structure using array-based structure.
2. Identify and discuss the different types of linked list structures.
3. Explain the concepts of recursion, stacks, and queues in data structures.
4. Create a Java program implementing the different types of linked lists, recursion, stacks and queues.

Lessons in this Module

Lesson 1: Linked Lists

Lesson 2: Recursion

Lesson 3: Stacks

Lesson 4: Queues

Lesson 1: Linked Lists

Learning Outcomes

- Explain the nature of linked lists in data structures.
- Perform the basic operations that can be done in a linked list.
- Appreciate the application of linked lists in real- world scenarios.

Introduction

Welcome to Lesson 1 of Module 2. In this lesson, we will be learning about linked lists. Let's get started.

Activity

Search about linked lists and express your idea through illustration. Include a short description of your illustration.

Analysis

Cite a real-life situation where linked list procedures are present.

Abstraction

Linked lists can be thought of from a high level perspective as being a series of nodes. In **DSA** our implementations of linked lists always maintain head and tail pointers so that insertion at either the head or tail of the list is a constant time operation. As such, linked lists in DSA have the following characteristics:

1. Insertion is $O(1)$
2. Deletion is $O(n)$
3. Searching is $O(n)$

1.1 Singly Linked List

Singly linked lists are one of the most primitive data structures you will find in this book. Each node that makes up a singly linked list consists of a value, and a reference to the next node (if any) in the list.

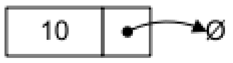

Figure 1.1: Singly linked list node

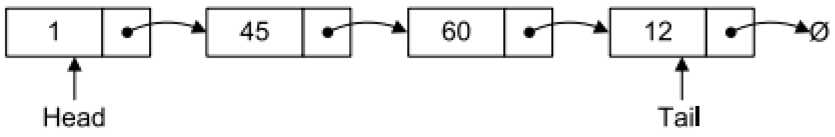

Figure 1.2: A singly linked list populated with integers

1.1.1 Insertion

In general when people talk about insertion with respect to linked lists of any form they implicitly refer to the adding of a node to the tail of the list. When you use an API like that of DSA and you see a general purpose method that adds a node to the list, you can assume that you are adding the node to the tail of the list not the head.

Adding a node to a singly linked list has only two cases:

1. *head* = ∅ in which case the node we are adding is now both the *head* and *tail* of the list; or
2. we simply need to append our node onto the end of the list updating the *tail* reference appropriately.

Pseudocode:

```
1)  algorithm Add(value)
2)      Pre: value is the value to add to the list
3)      Post: value has been placed at the tail of the list
4)      n ← node(value)
5)      if head = ∅
6)          head ← n
7)          tail ← n
8)      else
9)          tail.Next ← n
10)         tail ← n
11)     end if
12) end Add
```

As an example of the previous algorithm, consider adding the following sequence of integers to the list: 1, 45, 60, and 12, the resulting list is that of Figure 1.2.

1.1.2 Searching

Searching a linked list is straightforward: we simply traverse the list checking the value we are looking for with the value of each node in the linked list. The algorithm listed in this section is very similar to that used for traversal in Pseudocode 1.1.4.

Pseudocode:

```
algorithm Contains(head, value)
    Pre: head is the head node in the list
        value is the value to search for
    Post: the item is either in the linked list, true; otherwise false
    $n \leftarrow head$
    while $n \neq \emptyset$ and $n.\text{Value} \neq value$
        $n \leftarrow n.\text{Next}$
    end while
    if $n = \emptyset$
        return false
    end if
    return true
end Contains
```

1.1.3 Deletion

Deleting a node from a linked list is straightforward but there are a few cases we need to account for:

1. the list is empty; or
2. the node to remove is the only node in the linked list; or
3. we are removing the head node; or
4. we are removing the tail node; or
5. the node to remove is somewhere in between the head and tail; or
6. the item to remove doesn't exist in the linked list

The algorithm whose cases we have described will remove a node from anywhere within a list irrespective of whether the node is the *head* etc. If you know that items will only ever be removed from the *head* or *tail* of the list then you can create much more concise algorithms. In the case of always removing from the front of the linked list deletion becomes an $O(1)$ operation.

Pseudocode:

```
1)  algorithm Remove(head, value)
2)      Pre: head is the head node in the list
3)          value is the value to remove from the list
4)      Post: value is removed from the list, true; otherwise false
5)      if head = ∅
6)          // case 1
7)          return false
8)      end if
9)      n ← head
10)     if n.Value = value
11)         if head = tail
12)             // case 2
13)             head ← ∅
14)             tail ← ∅
15)         else
16)             // case 3
17)             head ← head.Next
18)         end if
19)         return true
20)     end if
21)     while n.Next ≠ ∅ and n.Next.Value ≠ value
22)         n ← n.Next
23)     end while
24)     if n.Next ≠ ∅
25)         if n.Next = tail
26)             // case 4
27)             tail ← n
28)         end if
29)         // this is only case 5 if the conditional on line 25 was false
30)         n.Next ← n.Next.Next
31)         return true
32)     end if
33)     // case 6
34)     return false
35) end Remove
```

1.1.4 Traversing the list

Traversing a singly linked list is the same as that of traversing a doubly linked list (defined in Figure 1.2). You start at the head of the list and continue until you come across a node that is ∅. The two cases are as follows:

1. *node* = ∅, we have exhausted all nodes in the linked list; or

2. we must update the *node* reference to be *node*.Next.

The algorithm described is a very simple one that makes use of a simple while loop to check the first case.

Pseudocode:

```
1)  algorithm Traverse(head)
2)      Pre: head is the head node in the list
3)      Post: the items in the list have been traversed
4)      n ← head
5)      while n ≠ 0
6)          yield n.Value
7)          n ← n.Next
8)      end while
9)  end Traverse
```

1.1.5 Traversing the list in reverse order

Traversing a singly linked list in a forward manner (i.e. left to right) is simple as demonstrated in Pseudocode 1.4.1. However, what if we wanted to traverse the nodes in the linked list in reverse order for some reason? The algorithm to perform such a traversal is very simple, and just like demonstrated in Pseudocode 1.3.1 we will need to acquire a reference to the predecessor of a node, even though the fundamental characteristics of the nodes that make up a singly linked list make this an expensive operation.

Pseudocode:

```
1)  algorithm ReverseTraversal(head, tail)
2)      Pre: head and tail belong to the same list
3)      Post: the items in the list have been traversed in reverse order
4)      if tail ≠ ∅
5)          curr ← tail
6)          while curr ≠ head
7)              prev ← head
8)              while prev.Next ≠ curr
9)                  prev ← prev.Next
10)             end while
11)             yield curr.Value
12)             curr ← prev
13)         end while
```

```
14)                 yield curr.Value
15)            end if
16)     end ReverseTraversal
```

This algorithm is only of real interest when we are using singly linked lists, as you will soon see that doubly linked lists (defined in Figure 1.5) make reverse list traversal simple and efficient, as shown in Fig 1.2.3.

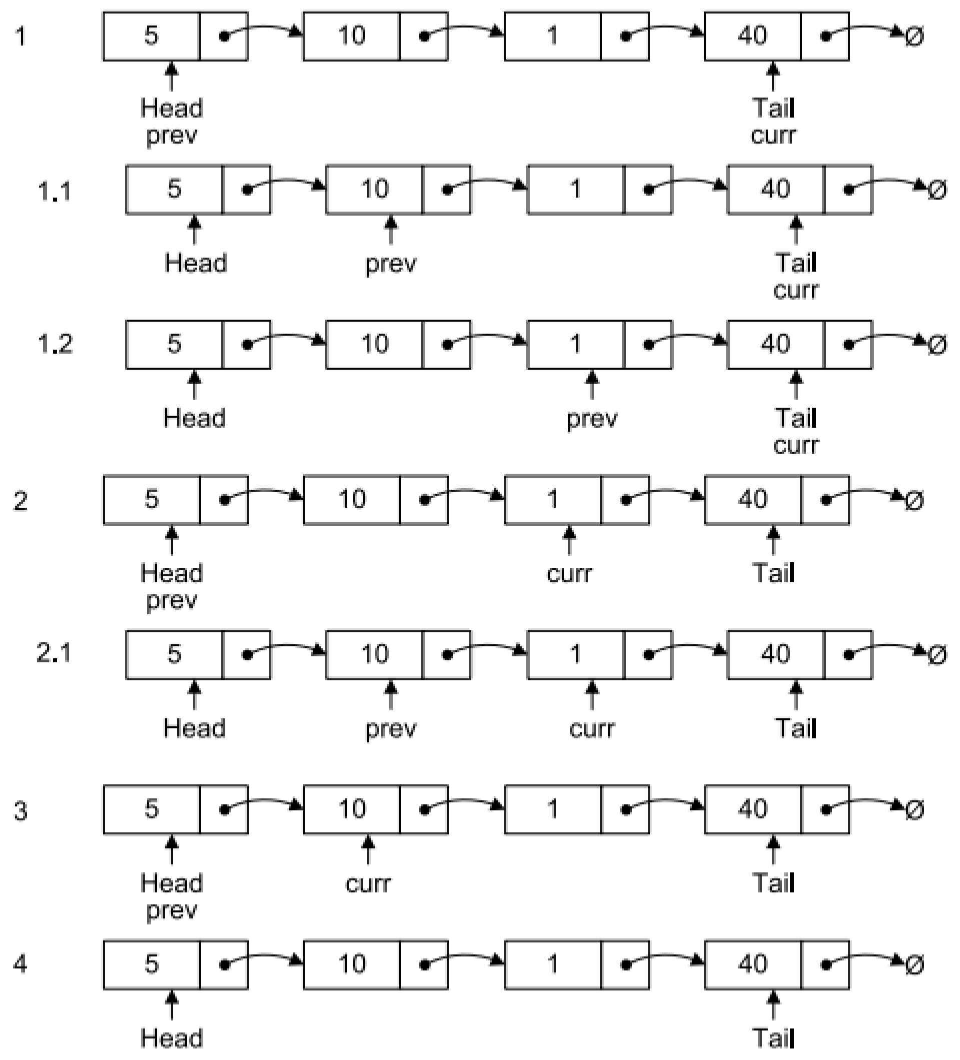

Figure 1.3: Reverse traversal of a singly linked list

1.2 Doubly Linked List

Doubly linked lists are very similar to singly linked lists. The only difference is that each node has a reference to both the next and previous nodes in the list.

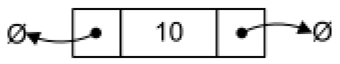

Figure 1.4: Doubly linked list node

The following algorithms for the doubly linked list are exactly the same as those listed previously for the singly linked list:

1. Searching (defined in Pseudocode 1.1.2)
2. Traversal (defined in Pseudocode 1.1.4)

1.2.1 Insertion

The only major difference between the algorithm in Pseudocode 1.1.1 is that we need to remember to bind the previous pointer of *n* to the previous tail node if *n* was not the first node to be inserted into the list.

Pseudocode:

```
1)  algorithm Add(value)
2)      Pre: value is the value to add to the list
3)      Post: value has been placed at the tail of the list
4)      n ← node(value)
5)      if head = ∅
6)          head ← n
7)          tail ← n
8)      else
9)          n.Previous ← tail
10)         tail.Next ← n
11)         tail ← n
12)     end if
13) end Add
```

Figure 1.5 shows the doubly linked list after adding the sequence of integers defined in Pseudocode 1.2.1.

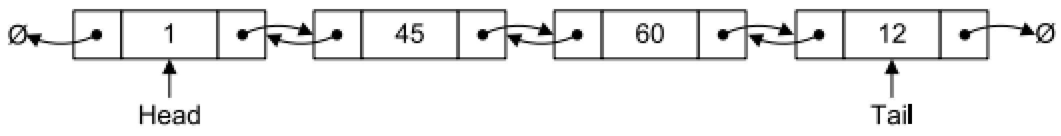

Figure 1.5: Doubly linked list populated with integers

1.2.2 Deletion

As you may of guessed the cases that we use for deletion in a doubly linked list are exactly the same as those defined in Pseudocode 1.1.3. Like insertion we have the added task of binding an additional reference (P revious) to the correct value.

Pseudocode:

```
algorithm Remove(head, value)
    Pre: head is the head node in the list
        value is the value to remove from the list
    Post: value is removed from the list, true; otherwise false
    if $head = \emptyset$
        return false
    end if
    if $value = head$.Value
        if $head = tail$
            $head \leftarrow \emptyset$
            $tail \leftarrow \emptyset$
        else
            $head \leftarrow head$.Next
            $head$.Previous $\leftarrow \emptyset$
        end if
        return true
    end if
    $n \leftarrow head$.Next
    while $n \neq \emptyset$ and $value \neq n$.Value
        $n \leftarrow n$.Next
    end while
    if $n = tail$
        $tail \leftarrow tail$.Previous
        $tail$.Next $\leftarrow \emptyset$
        return true
    else if $n \neq \emptyset$
        $n$.Previous.Next $\leftarrow n$.Next
        $n$.Next.Previous $\leftarrow n$.Previous
        return true
    end if
    return false
end Remove
```

1.2.3 Reverse Traversal

Singly linked lists have a forward only design, which is why the reverse traversal algorithm defined in Pseudocode 1.1.5 required some creative invention. Doubly linked lists make reverse traversal as simple as forward traversal (defined in Pseudocode 1.1.4) except that we start at the tail node and update the pointers in the opposite direction. Figure 1.6 shows the reverse traversal algorithm in action.

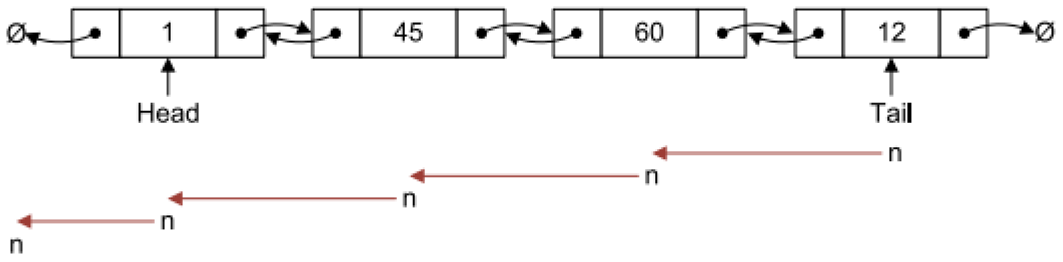

Figure 1.6: Doubly linked list reverse traversal

Pseudocode:

```
1)  algorithm ReverseTraversal(tail)
2)      Pre: tail is the tail node of the list to traverse
3)      Post: the list has been traversed in reverse order
4)      n ← tail
5)      while n ≠ Ø
6)          yield n.Value
7)          n ← n.Previous
8)      end while
9)  end ReverseTraversal
```

Application

Using java language, write a code for singly and doubly linked list with its characteristics.

Closure

Congratulations on completing Lesson 1. You can explore more about array-based structures in our next lesson.

Lesson 2: Recursion

Learning Outcomes

1. Explain the concept of recursion.
2. Understand the methods and the processes of recursion.
3. Appreciate the nature of recursion in solving recursion- related problems.

Introduction

Welcome back to array-based structures module. In this lesson, we will look at one of the important topics, "recursion". Let's get started.

Activity

Give at least two(2) examples/scenarios where you can apply recursion. You must show the steps and processes of how recursion is used or implemented in detail.

____
____
____
____

Analysis

On your own words, what is recursion and how it is applied to programming?

Abstraction

Any function which calls itself is called **recursive**. A **recursive** method solves a problem by calling a copy of itself to work on a smaller problem. This is called the **recursion step**. The **recursion step** can result in many more such recursive calls.

2.1 Why Recursion?

Recursion is a useful technique borrowed from mathematics. Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted. **Recursion** is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called the **base case**. The former, where the function calls itself to perform a subtask, is referred to as the **cursive case**. We can write all recursive functions using the format:

```
1.    if(test for the base case)
2.        return some base case value
3.    else if(test for another base case)
4.        return some other base case value
5.    //the recursive case
5.    else return (some work and then a recursive call)
```

As an example consider the factorial function: n! is the product of all integers between n and 1. The definition of recursive factorial looks like:

$$n! = 1, \qquad \textbf{if } n = 0$$

$$n! = n * (n\text{-}1)! \qquad \textbf{if } n > 0$$

This definition can easily be converted to recursive implementation. Here the problem is determining the value of n!, and the subproblem is determining the value of (n – l)!. In the recursive case, when n is greater than 1, the function calls itself to determine the value of (n –l)! and multiplies that with n. In the base case, when n is 0 or 1, the function simply returns 1. This looks like the following:

```
//calculates factorial of a positive integer
public int Factorial(int n){
        //base cases: fact of 0 is 1
        if(n == 0)
                return 1;
        //recursive case: multiply n by (n-1) factorial
        else
                return n*Factorial(n-1);
}
```

Now, let us consider our factorial function. The visualization of factorial function with n=4 will look like:

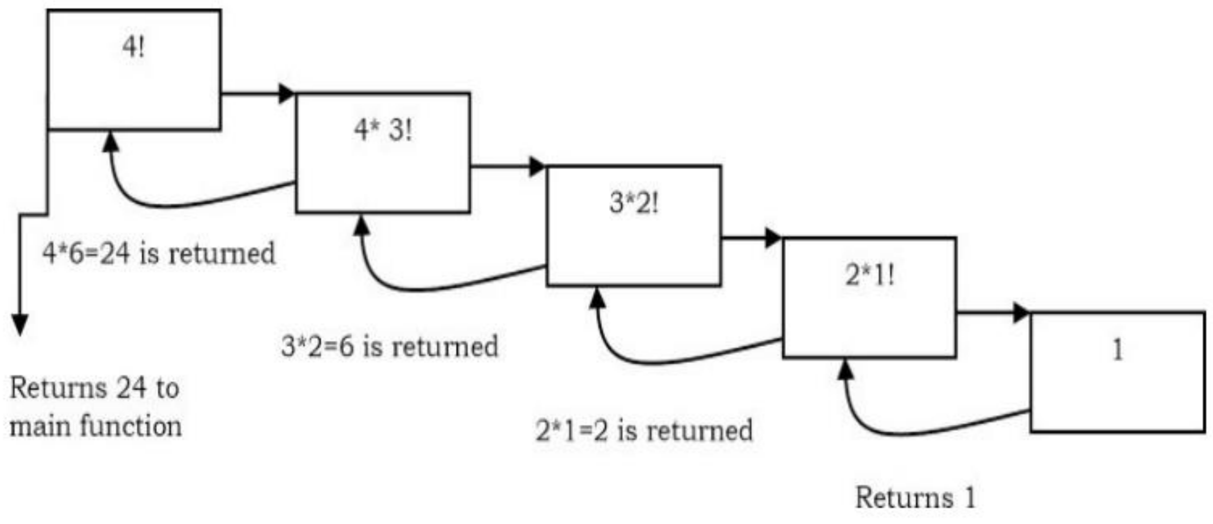

Figure 2.1: Visualizatino of factorial function (n=4)

2.2 Notes on Recursion

A recursive approach mirrors the problem that we are trying to solve. A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers.

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stackoverflow.
- Solutions to some problems are easier to formulate recursively
- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

Here are the list of recursion algorithms:

1. Fibonacci Series, Factorial Finding
2. Merge Sort, Quick Sort
3. Binary Search
4. Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder

5. Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
6. Dynamic Programming Examples
7. Divide and Conquer Algorithms
8. Towers of Hanoi
9. Backtracking Algorithms [we will discuss in next section]

Application

Search for a recursive mathematical algorithm and write its recursive programming algorithm counterpart using java language.

Closure

Congratulations on completing Lesson 2. You now know what recursion is and when to use it best.

Lesson 3: Stacks

Learning Outcomes

1. Explain the concepts of stacks.
2. Apply the methods associated with stack operations.
3. Appreciate stacks in solving stack- related problem in the real world.

Introduction

Welcome back to Module 2. In this lesson, we will be learning the processes of the stack. Let's get started.

Activity

List 5 real-life event/scenario which applies **stack** and explain briefly.

Analysis

In your own words, what is the advantage and disadvantages of **stack**?

Abstraction

A **stack** is a simple data structure used for storing data (similar to Linked Lists). In a **stack**, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a **stack**. The plates are added to the **stack** as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the **stack**. The first plate placed on the stack is the last one to be used.

A **stack** is an ordered list in which insertion and deletion are done at one end, called **top**. The last element inserted is the first one to be deleted. Hence, it is called the **Last in First out** (LIFO) or **First in Last out** (FILO) list.

Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called **push**, and when an element is removed from the stack, the concept is called **pop**. Trying to pop out an empty stack is called **underflow** and trying to push an element in a full stack is called **overflow**.

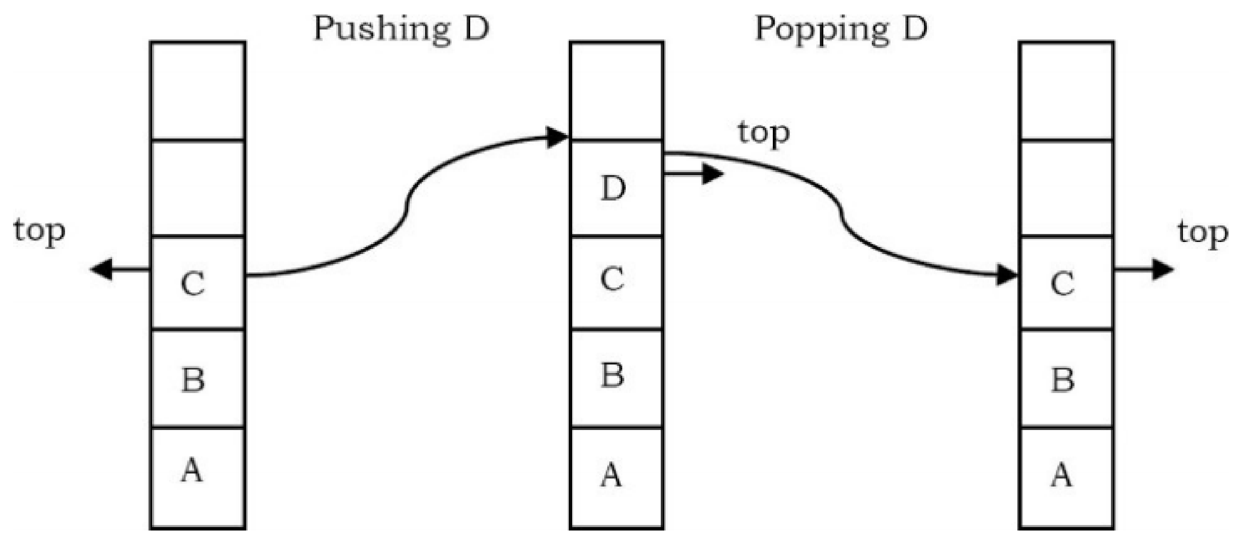

Figure 3.1: Changing stack values (push and pop)

3.1 Stack Operations

The following operations make a stack an ADT. For simplicity, assume the data is an integer type.

Main stack operations

1. Add an element onto the top of the stack. **void push(***int data***)**
2. Access the current element on the top of the stack. **int Top()**
3. Remove the current element on the top of the stack. **int pop()**

These three operations are usually named **push**, **peek**, and **pop**, respectively.

In the Stack, operations **pop** and **peek** cannot be performed if the stack is empty. Attempting the execution of **pop** or **peek** on an empty stack throws an **exception**. Trying to push an element in a full stack throws an **exception**. This exceptions are error conditions throwed by an operation that cannot be executed.

Here are some of the applications in which stacks play an important role.

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to Problems section)
- Page-visited history in a Web browser [Back Buttons]

- Undo sequence in a text editor
- Matching Tags in HTMLand XML

Indirect applications

- Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)
- Component of other data structures (Example: Simulating queues, refer Queues chapter)

3.2 Implementation

There are many ways of implementing stack; below are the commonly used methods.

1. Simple array based implementation
2. Linked lists implementation

Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.

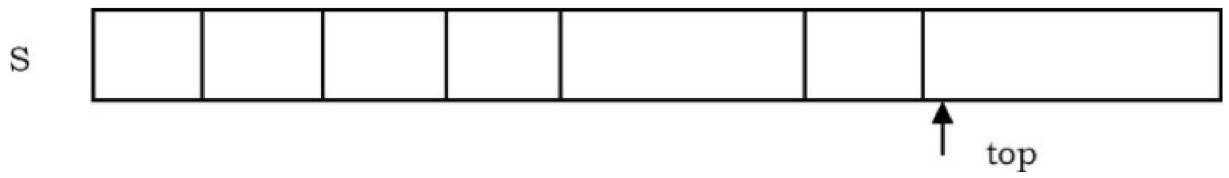

Figure 3.2: Stack in a simple array

The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception*.

Simple Array Implementation Performance and Limitation

Performance: Let *n* be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of push()	O(1)
Time Complexity of pop()	O(1)
Time Complexity of size()	O(1)
Time Complexity of isEmpty()	O(1)
Time Complexity of isFullStack()	O(1)
Time Complexity of deleteStack()	O(1)

Figure 3.3: Simple Array Implementation Performance and Limitation

Limitations: The maximum size of the stack must first be defined and it cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

Linked List Implementation

The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).

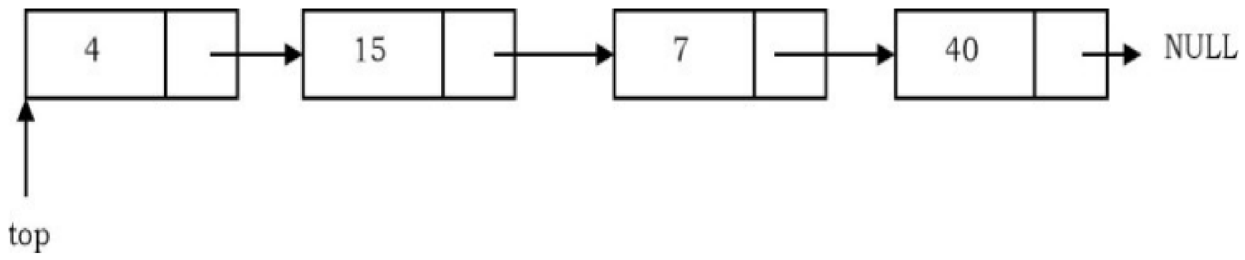

Figure 3.4: Stack in a linked list

Linked List Implementation Performance and Limitations

Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of create Stack: DynArrayStack()	O(1)
Time Complexity of push()	O(1) (Average)
Time Complexity of pop()	O(1)
Time Complexity of top()	O(1)
Time Complexity of isEmpty()	O(1)
Time Complexity of deleteStack()	O(*n*)

Figure 3.5: Linked List Implementation Performance and Limitations

Array Implementation & Linked List Implementation Points

Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) - "amortized" bound takes time proportional to n..

Linked List Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time O(1).

Application

List 3 system processes which implements **stack** algorithm.

Closure

Congratulations on completing Lesson 4. You are now equipped with the knowledge about stacks.

Lesson 4: Queues

Learning Outcomes

1. Define and explain queues as one of the array- based structures.
2. Understand the methods and operations that can be done in queues.
3. Appreciate queues and its concepts in data structures.

Introduction

Welcome back to Module 2. In this lesson, we will be learning about the processes of queues. Let's get started.

Activity

List 5 real-life event/scenario which applies **queue** and explain briefly.

Analysis

In your own words, what is the advantage and disadvantages of **queue**?

Abstraction

Queues are an essential data structure that are found in vast amounts of software from user mode to kernel mode applications that are core to the system. Fundamentally they honour a **first in first out** (FIFO) strategy, that is the item first put into the queue will be the first served, the second item added to the queue will be the second to be served and so on.

A traditional queue only allows you to access the item at the front of the queue; when you add an item to the queue that item is placed at the back of the queue.

Historically queues always have the following three core methods:

Enqueue: places an item at the back of the queue;

Dequeue: retrieves the item at the front of the queue, and removes it from the queue;

Peek: retrieves the item at the front of the queue without removing it from the queue

As an example to demonstrate the behaviour of a queue we will walk through a scenario whereby we invoke each of the previously mentioned methods observing the mutations upon the queue data structure. The following list describes the operations performed upon the queue in Figure 6.1:

1. Enqueue(10)
2. Enqueue(12)
3. Enqueue(9)
4. Enqueue(8)
5. Enqueue(3)
6. Dequeue()
7. Peek()
8. Enqueue(33)
9. Peek()
10. Dequeue()

4.1 A standard queue

A queue is implicitly like that described prior to this section. In DSA we don't provide a standard queue because queues are so popular and such a core data structure that you will find pretty much every mainstream library provides a queue data structure that you can use with your language of choice. In this section we will discuss how you can, if required, implement an efficient queue data structure.

The main property of a queue is that we have access to the item at the front of the queue. The queue data structure can be efficiently implemented using a singly linked list (defined in Figure 1.1). A singly linked list provides *O*(1) insertion and deletion run time complexities. The reason we have an *O*(1) run time complexity for deletion is because we only ever remove items from the front of queues (with the Dequeue operation). Since we always have a pointer to the item at the head of a singly linked list, removal is simply a case of returning the value of the old head node, and then modifying the head pointer to be the next node of the old head node. The run time complexity for searching a queue remains the same as that of a singly linked list: *O*(n).

4.2 Priority Queue

Unlike a standard queue where items are ordered in terms of who arrived first, a priority queue determines the order of its items by using a form of custom comparer to see which item has the highest priority. Other than the items in a priority queue being ordered by priority it remains the same as a normal queue: you can only access the item at the front of the queue.

A sensible implementation of a priority queue is to use a heap data structure (defined in Module 3 Lesson 4). Using a heap we can look at the first item in the queue by simply returning the item at index 0 within the heap array. A heap provides us with the ability to construct a priority queue where the items with the highest priority are either those with the smallest value, or those with the largest.

4.3 Double Ended Queue

Unlike the queues we have talked about previously in this chapter a double ended queue allows you to access the items at both the front, and back of the queue. A double ended queue is commonly known as a deque which is the name we will here on in refer to it as.

A deque applies no prioritization strategy to its items like a priority queue does, items are added in order to either the front of back of the deque. The former properties of the deque are denoted by the programmer utilising the data structures exposed interface.

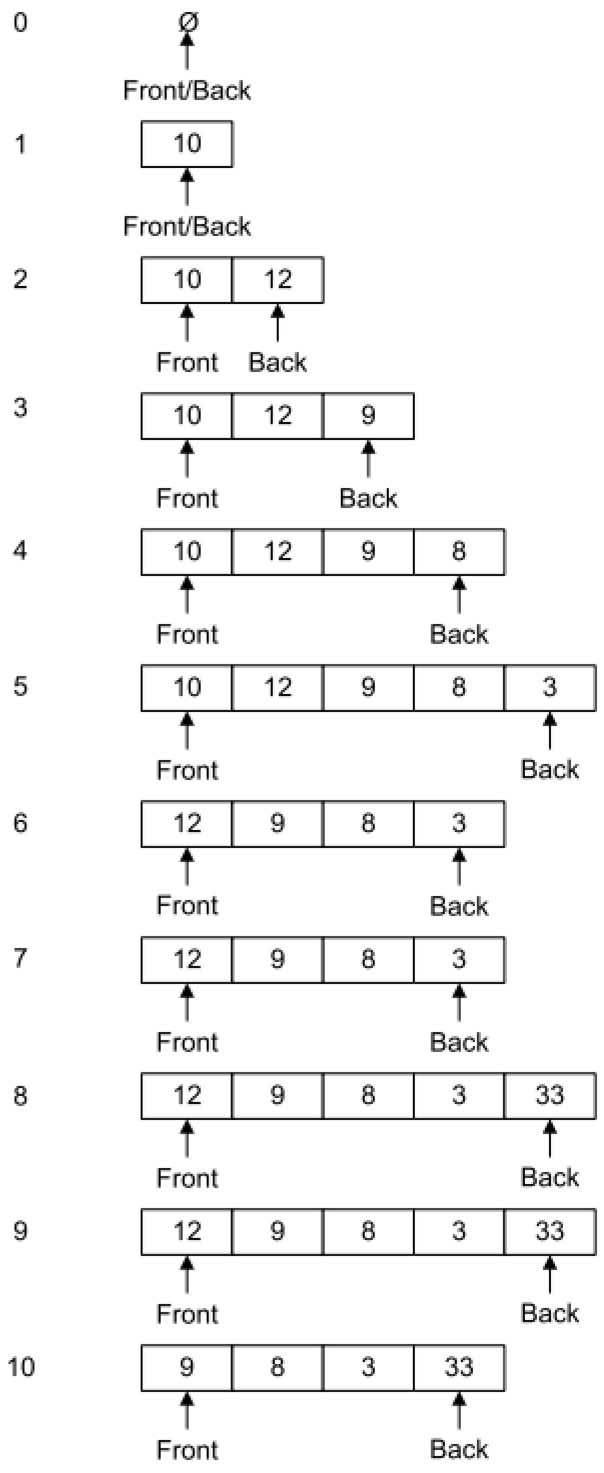

Figure 4.1: Queue mutations

Deque's provide front and back specific versions of common queue operations, e.g. you may want to enqueue an item to the front of the queue rather than the back in which case you would use a method with a name along the lines of *EnqueueFront*. The following list identifies operations that are commonly supported by deque's:

- EnqueueFront
- EnqueueBack
- DequeueFront
- DequeueBack
- PeekFront
- PeekBack

Figure 6.2 shows a deque after the invocation of the following methods (inorder):

1. EnqueueBack(12)
2. EnqueueFront(1)
3. EnqueueBack(23)
4. EnqueueFront(908)
5. DequeueFront()
6. DequeueBack()

The operations have a one-to-one translation in terms of behavior with a normal queue or priority queue. In some cases, the set of algorithms that add an item to the back of the deque may be named as they are with normal queues, e.g. *EnqueueBack* may simply be called *Enqueue* and so on. Some frameworks also specify explicit behavior's that data structures must adhere to. This is certainly the case in .NET where most collections implement an interface that requires the data structure to expose a standard *Add* method. You can safely assume that the Add method will simply enqueue an item to the dequeue's back in such a scenario.

Concerning algorithmic run time complexities, a dequeue is the same as a normal queue. Enqueueing an item to the back of the queue is $O(1)$; additionally enqueuing an item to the front of the queue is also an $O(1)$ operation.

A dequeue is a wrapper data structure that uses either an array or a doubly-linked list. Using an array as the backing data structure would require the programmer to be explicit about the array's size up front. This would provide an obvious advantage if the programmer could deterministically state the maximum number of items the deque would contain at any one time. Unfortunately, in most cases, this doesn't hold; thus, the backing array will inherently incur the expense of invoking a resizing algorithm that would most likely be an $O(\text{n})$ operation. Such an approach would also leave the library developer to look at array minimization techniques as well; it could be that after several invocations of the resizing algorithm and various mutations on the deque later that we have an array taking up a considerable amount of memory yet we are only using a few small percentage of that memory. An algorithm described would also be O(n), yet its invocation would be harder to gauge strategically.

To bypass all the issues mentioned above a dequeue typically uses a doubly linked list as its baking data structure. While a node that has two pointers consumes more memory than its array item counterpart it makes redundant the need for expensive resizing algorithms as the data structure increases in size dynamically. With a language that targets a garbage collected virtual machine memory reclamation is an opaque process as the nodes that are no longer referenced become unreachable and are thus marked for collection upon the next invocation of the garbage collection algorithm. With C++ or any other language that uses explicit memory allocation and deallocation it will be up to the programmer to decide when the memory that stores the object can be freed.

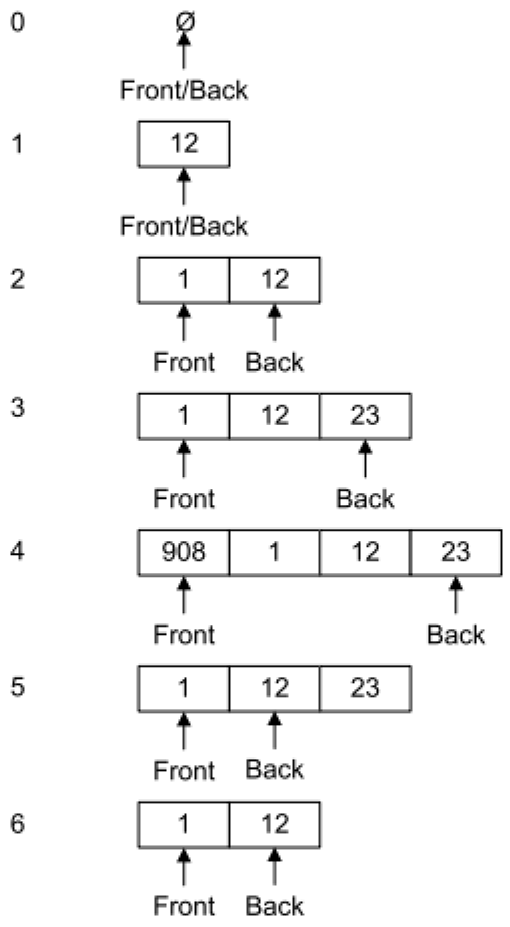

Figure 4.2: Deque data structure after several mutations

Application

List 3 system processes which implements **queue** algorithm.

Closure

Congratulations on completing Lesson 4. You are now equipped with the knowledge about queues.

Module Assessment

1. Using java language, write a program that performs all the stack operations.
2. Using java language, write a program that performs all the queue operations.

Module Summary

- Linked lists are good to use when you have an unknown number of items to store.
- You should also use linked lists when you will only remove nodes at either the head or tail of the list to maintain a constant run time.
- Singly linked lists should be used when you are only performing basic insertions.
- Doubly linked lists are more accommodating for non-trivial operations on a linked list.
- Use a doubly linked list when you require forwards and backwards traversal.
- Queues can be ever so useful; for example the Windows CPU scheduler uses a different queue for each priority of process to determine which should be the next process to utilise the CPU for a specified time quantum.
- A recursive method solves a problem by calling a copy of itself to work on a smaller problem.
- Recursion is a useful technique borrowed from mathematics.
- Recursive code is generally shorter and easier to write than iterative code.
- Recursion is most useful for tasks that can be defined in terms of similar subtasks.
- A recursive function performs a task in part by calling itself to perform the subtasks.
- A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers.
- A stack is an ordered list in which insertion and deletion are done at one end, called top.
- When an element is inserted in a stack, the concept is called push, and when an element is removed from the stack, the concept is called pop.
- Normal queues have constant insertion and deletion run times.
- Searching a queue is fairly unusual—typically you are only interested in the item at the front of the queue.

- Queues are a very natural data structure, and while they are fairly primitive they can make many problems a lot simpler.

References

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHILearning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), DreamtechPress