Journal of Faculty of Engineering & Technology

**Journal homepage:** [**www.pu.edu.pk/journals/index.php/jfet/index**](http://www.pu.edu.pk/journals/index.php/jfet/index)

[**Learning Stack: Structured, Object Oriented, Generic and Design Pattern Approach**](http://sebastian.doc.gold.ac.uk/papers/Language_Learning/NNPDA_App.pdf)

Muhammad Shoaib Farooq1,3, Aqsa Ali1, Kamran Abid2, Adnan Abid1

1Department of Computer Science, University of Management and Technology, Pakistan.

2College of Engineering and Emerging Sciences, University of the Punjab, Lahore, Pakistan.

3Abdul Wali Khan University, Mardan, Pakistan.

{Shoaib.farooq; aqsa.ali; adnan.abid}@umt.edu.pk, kamran@pu.edu.pk

**Abstract**

The concept of a stack, its usage and implementation hold central importance in computer programming courses taught in electrical and computer system engineering programs, as well as in computer science discipline. Stack covers all types of computational problem that can be solved with *Last-In-First-Out* (LIFO) principle. In the presence of a conventional course outlines and literature, there is a strong need to define standard topics and relevant subtopics for teaching stack. In this article, we highlight and address this problem, and as a principal contribution we define a taxonomy of stack from the perspective of its implementation using structured, and object oriented programming paradigms, and have also discussed its implementation using design patterns. To this end, we have used C++ language to present the code for different operations, and have supported the explanation with the help of memory diagrams, complexity issues, and example usage. We have also discussed the formal definition of stack.

***Keywords****:* stack; data structure; stack design patterns; formal methods;

**1. Introduction.** A stack is an ordered collection of the same type of elements in which insertion and deletion are made at one end called top, based on LIFO principle ‎[1]‎[2]‎[3] . It means the very first item that was inserted shall be removed at the end. The stack is considered among the core data structures and is useful in many applications e.g. Redo logs, arithmetic expression evaluation by enforcing precedence and associative rules, depth first search, conserving memory during function calls for local variables etc. The concept of stack is a core part of electrical, computer system engineering programs, and computer science curricula ‎[6]‎[7]‎[8] . There are two major approaches for teaching computer programming courses. One approach is referred to as an imperative first approach, whereas the other is known as object first approach. Universities that follow imperative first approach use procedural (structured) techniques in the data structure course, while the universities that follow object first approach use object oriented abstract data type based approach for implementing data structures ‎[13] . In the course outlines of data structure stack is usually presented at a high abstraction level, and the details are left for the faculty members. Therefore, in practice, the relevant subtopics totally depend upon the faculty consensus and as a result we have different variants of teaching this topic. To the best of our knowledge, no concrete and well defined subtopics on the stack have been defined.

The major focus of this article is to figure out all possible implementations of stack using structured, object oriented, generic and design pattern approaches. These approaches lead us to propose a taxonomy of the implementation of stack. The other contribution of this work is that we have implemented all possible variants of stack and illustrate their codes with the help of memory diagrams. We have used C++ for implementation of the stack, because this language has been commonly used as a first programming language in engineering and computer science disciplines, and it supports both imperative first, and object first teaching approaches in an equally good manner.

The rest of the article has been organized in such a way that Section 2 presents the related work, whereas Section 3 presents the taxonomy in detail. Furthermore, in Section 4, we present the relative importance of each variant in terms of teaching the course of data structures. Lastly, we present the conclusion of this work in Section 5.

**2. Related Work**

The concept of stack was introduced by Alan M. Turing in 1946 in the design of computer using subroutines. Alan M. Turing used the term bury and unbury for adding a data on the top of the stack, and removing data from it ‎[18]. Currently, push and pop operations are being used as the names of these operations. Gelfand at. el. propose design pattern for the stack data structure, and highlighted its importance in the curriculum‎ [9]. Niculescu discussed the importance of design patterns in data structure implementation ‎[11]. The ACM curriculum 2001, 2008 and 2013 proposes contents related to stack in CS1 and CS2 course, but detail sub topics have not been included ‎[14]‎[15]‎[16]. There are two major variants of the course of data structures which discuss stack as a core topic and are named as *Fundamental Data Structures* and *Data Structures and Algorithms*.

**3. Classification of Stack**

In this section firstly we present a formal definition of stack, which is then followed by a taxonomy of stacks based on its implementation using structured and object oriented paradigms. Figure 1 shows the defined taxonomy with all details.

**3.1 Formal Definition of a Stack**

As an abstract definition stack is defined as a linear list of items in which all additions and deletion are restricted to one end that is *top* ‎[10]‎[17]‎[19] . Mathematicians formally solve all types of computational problems using three data structures i.e. *set*, *sequence* and *map*. A *sequence* is an ordered collection of objects, and stack is also an ordered list based on last-in-first-out (LIFO) principle. Hence, sequence is considered to be a better suitable collection of modeling a stack. The formal specification of stack using [Vienna Development Method](http://en.wikipedia.org/wiki/Vienna_Development_Method) (VDM) has been presented in Table 1. Here, *Element= TOKEN* mean*s* stack can store any type just like templates in C++. The *init mk-Stack* is initialization function that shows initially the stack is empty. The *ext wr* in push function means write/read access on the stack. There is no precondition for *push* function, whereas the post condition ensures that the new state of the stack should be equal to parameter *p* concatenated with the old state of the stack. The *pop* method has read and write access on the stack. The pre-condition ensures stack should not be empty before *pop,* whereas the post-condition ensures two things: firstly, new state of stack should be equal to the *tail* of stack (in sequence tail means all except head), and secondly, *itemDeleted* should be equal to the *head* of old state of stack. Here, stack with header line means a previous state of stock. The *isEmpty* function returns *true* if the stack sequence is empty, and *false* otherwise.

|  |
| --- |
| **Table 1 : Formal Specification of Stack in VDM** |
| **types** *Element = TOKEN***state** *Stack* **of** *stack : Element\**  **init mk-***Stack(s)∆***s =[]** **end***push(p : Element)***ext wr** *stack : Element\****pre** *TRUE***post** *stack= [p] ^* *pop() itemDeleted : Element* **ext wr** *stack : Element\****pre** stack ≠ []**post** *stack =* ***tl***  *^ itemDeleted = hd* *isEmpty() query :β***ext rd** *stack : Element\****pre** *TRUE***post** *query ⇔ stack = []* |

**3.2 Taxonomy of Stack using Different Implementations**

We have defined two major classes based on the implementation of stack using two different programming paradigms, structured paradigm and object oriented programming paradigm. The structured paradigm supports imperative first approach of teaching computer programming. Whereas, the object oriented approach can be incorporated using both imperative first and object first approaches. However, in the former case we need to teach the course of object oriented programming before teaching data structures.

The stack can be implemented using two different core data structures called array and linked lists. We have discussed all different variants of stacks irrespective of programming paradigms using these core data structures. The taxonomy tree shows that structured approach offers the aforementioned two different ways of implementing stack i.e. by using arrays, and by using linked lists. Whereas, the object oriented approach offers many different abstractions while using the same basic data structures of arrays and linked lists. The taxonomy tree presented in Figure 1 clearly shows that the object oriented implementation is richer than the relatively simplified implementations using structured programming. In all the object oriented approaches, we can implement different ADTs using array and linked list. Furthermore, in the same paradigm we can implement *generic* implementations using both array and linked list. Lastly, we have discussed the object oriented implementation of stack using *adapter* design pattern, where we implement it in two different ways while using the concepts of inheritance and composition.



Figure 1: Taxonomy of stack based on its implementation

Implementation of stack offers all major services *push*, *pop*, and *peek* which are implemented in the form of independent global functions outside the client program ‎[17]. These stack services can be implemented with the help of fixed sized arrays, as well as using dynamic sized linked list ‎[20] . Apart from the major services there is a supporting method known as *isEmpty* that helps identifying if the stack possesses any data or not.

The array based implementation involves fixed length stack and therefore requires a method *isFull* that informs the *main* program that the stack possesses maximum allowed objects or not. Whereas, in the case of linked list based implementation the stack can grow and shrink to any limit, hence *isFull* method is not required. The signatures of all the methods have been presented in Table 2.

**Table 2: Signatures of major services/methods implemented in the Stack**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Purpose** | **Used in Linked List** | **Used in Array** | **Signature** |
| Push | Add an element in the stack | Yes | Yes | bool push(int p) (for array) |
| void push(int p) (for linked list) |
| Pop | Remove an element from the stack | Yes | Yes | bool pop(int &p) |
| Peek | Get value of the element at the tos | Yes | Yes | Bool peek(int &p) |
| isEmpty | Tells if the stack is empty or not | Yes | Yes | Bool isEmpty() |
| isFull | Tells if the stack is full (based on its size) | No | Yes | Bool is Full() |

In order to better understand each variant, it is pertinent to note that for each array based implementation, we have discussed *push*, *pop*, *peek*, *isFull*, and *isEmpty* methods, whereas for linked list based implementations we exclude *isFull* as it is no more valid. We further discuss each implementation using a case study example in the client program (*main* function) which swaps two variables using stack. We have also presented a memory diagram for each step of the swapping function for array and linked list.

**3.2.1 Structured Implementation of Stack**

*3.2.1.1 Array based (fixed size) stack***:** Implementation of stack using an array is a simple and straightforward technique in which the size of a stack cannot be changed after its creation. The *push* operation may or may not be successful as it depends on the size of stack ‎[4]‎[5] . The return type of the *push* function while implementing stack using an array is *a Boolean* as shown on line 4 of Code Listing 1. This function returns *true* if the inserted data has been successfully pushed on the top of the stack. Whereas, it returns *false* in case the array is already full. A memory variable Top of Stack (*tos*) is used as index to support *push* and *pop* operations on the stack. The initial value of this variable is -1 which means the stack is empty. Every *push* operation places data on the array after incrementing in *tos* variable by one. Similarly, every *pop* operation of stack decrements *tos* variable by one after placing data form array ‘a’ to parameter ‘p’ shown in the signatures presented in Table 1. The *peek* operation is used to show the top value of a stack, and is the same as that of *pop,* except it does not change the variable *tos*. The *main* function in Code Listing 1 swaps two variables with *push* and *pop* operations implemented using C++, whereas, the memory diagrams are shown in Figure 2.

****

Figure 2: Array based (fixed stack) size (code listing 1). (a) Initial empty stack (line no 3). (b) Push 10 as element onto stack with tos=0 (line no 35). (c) Push 20 as element onto stack with tos=1 (line no 36). (d) Pop 20 as an element from stack with tos=0 (line no 38).

|  |
| --- |
| Code Listing 1: Array based stack in C++ |
| 1. **const int** size=10;
2. **int** a[size];
3. **int** tos=-1;
4. **bool** push(**int** p) {
5. **bool** flag=true;
6. **if**(tos!= size)
7. flag=**false**;
8. **else**{
9. tos++;
10. a[tos]=p;
11. }
12. **return** flag;
13. }
14. **bool** pop (**int** &p){
15. **bool** flag=true;
16. **if**(tos==-1)
17. flag=false;
18. **else**{
19. p=a[tos];
20. tos--;
21. }
22. **return** flag;
23. }
 | 1. **bool** peek (**int** &p){
2. **bool** flag=**true**;
3. **if**(tos==-1){
4. flag=**false**;
5. }
6. **else**{
7. p=a[tos];
8. }
9. **return** flag;
10. }
11. **void** main (){
12. **int** a=10;
13. **int** b=20;
14. cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;
15. push(a);
16. push(b);
17. pop(a);
18. pop(b);
19. cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;
20. }
 |

*3.2.1.2 Linked list based (dynamic size) stack:* Implementation of stack using a linked list is another straight forward technique in which the size of a stack can grow and shrink dynamically while using *push* and *pop* operations, respectively. The *push* operation is always successful if space is available in the heap memory. Here, return type of *push* operation is *void* as shown in Table 1.

The variable *tos* (Top of stack) is the head pointer of the linked list, and is used for *push* and *pop* operations. The initial value of this variable is NULL that represents an empty stack. Every *push* operation adds a new node on the front of the linked list, and places the new data in that node. It also sets the newly inserted node as *tos*.

Similarly, every *pop* operation of stack removes the head node form the linked list and returns the data in a reference variable. The *peek* operation is same as that of *pop* operation, except it does not remove the head node. The *main* method in Code Listing 2 swaps two variables with *push* and *pop* operations. Whereas, Figure 3 shows all the steps of the *main* method with the help of memory diagrams.

****

Figure 3: Linked list based (dynamic stack) size (code listing 2). (a) Initial stack (b) push value of ‘a’ on stack (c) push value of ‘b’ on stack (d) pop value of ‘a’ from stack (e) pop value of ‘b’ from stack.

|  |
| --- |
| Code Listing 2: Linked list based stack in C++ |
| 1. **struct** Node{
2. **int** data;
3. Node \*next;
4. } \*tos;
5. **bool** isEmpty(){
6. **if** (tos==NULL)
7. **return** **true**;
8. **else**
9. **return** **false**;
10. }
11. **void** push(**int** d){
12. **if**(isEmpty()){
13. tos=**new** Node;
14. tos->data=d;
15. tos->next=NULL;
16. }
17. **else**{
18. Node \*temp;
19. temp=**new** Node;
20. temp->data=d;
21. temp->next=tos;
22. tos=temp;
23. }
24. }
25. **bool** pop(**int** &d){
26. **bool** flag=true;
27. **if**(!isEmpty()){
28. Node \*temp;
29. d=tos->data;
 | 1. temp=tos;
2. tos=tos->next;
3. **delete** temp;
4. }
5. **else**
6. flag=**false**;
7. **return** flag;
8. }
9. **bool** peek(**int** &d){
10. **bool** flag=true;
11. **if**(!isEmpty()){
12. d=tos->data;
13. }
14. **else**
15. flag=**false**;
16. **return** flag;
17. }
18. **void** main (){
19. **int** a=10;
20. **int** b=20;
21. cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;
22. push(a);
23. push(b);
24. pop(a);
25. pop(b);
26. cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;
27. }
 |

**3.2.2 Object Oriented Implementation of Stack**

A stack can be implemented using object oriented programming in the form of an Abstract Data Type (ADT) ‎[4]. This implementation involves classes which require a name, data members, and methods. Figure 4 shows UML diagram of five basic operations of stack ADT. The Code Listing 3 shows array based, while Code Listing 4 shows linked list based ADT of implementing a stack in C++. Application wise, the ADT based implementation of stack provides advantage of reusability and maintenance of data and functions.

The methods remain the same as shown in Table 1. However, there are different data members while implementing stack using array and using linked list. In the array based implementation as shown in Code Listing 3, we need an array to store the data in a stack, and the limitation with the array is its size which requires us to use yet another data member *size*. The third data member is *tos* variable which is an integer and holds a value -1 at the start, showing that the stack is empty. It increases by one with each successful *push* operations, and decreases by 1 with each successful *pop* operation.



Figure 4: UML diagram of Stack Class

|  |
| --- |
| **Code Listing 3: ADT stack (array based) in C++** |
| 1 **class** Stack{2 **public:**3 Stack(**int** s){size=s; 4 data=**new** **int**[size];5 tos=-1;6 }7 **bool** isEmpty(){ **return** tos==-1; }8 **bool** isFull(){**return** tos==size-10;}9 **bool** push(**int** d){10 **bool** flag=**true**;11 **if** (!isFull()){12 tos++;13 data[tos]=d;14 }15 **else**16 flag= **false**;17 **return** flag;18 }19 **bool** pop(**int** &d){20 if (!isEmpty()){21 d=data[tos];22 tos--;23 **return** **true**;24 } **else**25 **return** false; 26 }27 **bool** peek (**int** &p){28 **bool** flag=**true**;29 **if**(tos==-1){ | 30 flag=**false**;31 }32 **else**{33 p=a[tos];34 }35 **return** flag;36 }37 **private**:38 **int** \*data;39 **int** size;40 **int** tos;41 };42 void main (){43 **int** a=10;44 **int** b=20;45 cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;46 Stack s(2);47 s.push(a);48 s.push(b);49 s.pop(a);50 s.pop(b);51 cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;52 } |

The linked list based implementation of stack involves a Node class that defines a storage space of a data value along with a pointer that connects one data element (node) with its subsequent data element (node). The stack class holds a pointer of type Node that points to the first node of the stack, and is shown as *tos* in the Code Listing 4. The *push* function adds a new node at the beginning of the stack and populates its data with the input value. It also sets the new node as *tos*, and connects it to the previously existing *tos*. Whereas, the *pop* function removes the top of the stack, in case it is not empty, and then it sets the subsequent node to be the *tos*.

**3.2.3 Generic stack**

Generic stacks help storing the data of any specific type in the stack. C++ supports such flexibility with the help of template classes. There is an arbitrary data type hard coded in class stack and it should be decided at an object declaration time by passing type parameter. The UML diagram shown in Figure 5 shows stack operations with generic parameter. The Code Listings 5 and 6 show array and linked list based generic stack in C++, respectively.



Figure 5: UML diagram of Stack Generic class

|  |
| --- |
| **Code Listing 4: ADT stack (Linked list based) in C++** |
| 1 **class** Node{2 **public:**3 **int** data;4 Node \*next;5 };6 **class** Stack{7 **public:**8 Stack(){9 tos=NULL;10 }11 **bool** isEmpty(){12 **if**(tos==NULL)13 **return** true;14 **else**15 **return** false;16 }17 **void** push(**int** d){18 **if**(isEmpty()){19 tos=**new** Node;20 tos->data=d;21 tos->next=NULL;22 }23 **else**{24 Node \*temp;25 temp=**new** Node;26 temp->data=d;27 temp->next=tos;28 tos=temp;29 }30 }31 **bool** pop(**int** &d){32 **bool** flag=true;33 **if**(!isEmpty()){ | 34 Node \*temp;35 d=tos->data;36 temp=tos;37 tos=tos->next;38 **delete** temp;39 }40 **else** flag=**false**;41 **return** flag;42 }43 **bool** peek(**int** &d){44 **bool** flag=**true**;45 **if**(!isEmpty()){46 d=tos->data;47 }48 **else** flag=**false**;49 **return** flag;50 }51 **private**:52 Node \*tos;53 };54 **void** main (){55 **int** a=10;56 **int** b=20;57 cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;58 Stack s;59 s.push(a);60 s.push(b);61 s.pop(a);62 s.pop(b);63 cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;64 } |

|  |
| --- |
| **Code Listing 5: Generic stack (array based) in C++** |
| 1 **template** <**class** T>2 **class** Stack{3 **public:**4 **Stack**(**int** s){size=s; 5 data=**new** T[size];6 tos=-1;7 }8 **bool** isEmpty(){**return** tos==-1; }9 **bool** isFull(){**return** tos==size-1;}10 **bool** push(T d){11 **bool** flag=**true**;12 **if** (!isFull()){13 tos++;14 data[tos]=d;15 }16 **else**17 flag= false;18 **return** flag;19 }20 **bool** pop(T &d){21 **if** (!isEmpty()){22 d=data[tos];23 tos--;24 **return** **true**;25 }**else**26 **return** **false**; 27 }28 **bool** peek (T &p){ | 29 **bool** flag=**true**;30 **if**(tos==-1){31 flag=**false**;32 }33 **else**{34 p=a[tos];35 }36 **return** flag;37 }38 **private**:39 T \*data;40 **int** size;41 **int** tos;42 };43 **void** main (){44 **int** a=10;45 **int** b=20;46 cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;47 Stack <**int**> s(2);48 s.push(a);49 s.push(b);50 s.pop(a);51 s.pop(b);52 cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;53 } |

|  |
| --- |
| **Code Listing 6: Generic stack (linked list based) in C++** |
| 1. **template** <**class** T>
2. **class** Node{
3. **public**:
4. T data;
5. Node \*next;
6. };
7. **template** <**class** T>
8. **class** Stack{
9. **public:**
10. Stack(){tos=NULL;}
11. **bool** isEmpty(){
12. **if**(tos==NULL)
13. **return** **true**;
14. **else**
15. **return** **false**;
16. }
17. **void** push(Node\* d){
18. **if**(isEmpty()){
19. tos=**new** Node;
20. tos=d;
21. }
22. **else**{
23. d->next=tos;
24. tos=d;
25. }
26. }
27. **bool** pop(Node\* d){
28. **bool** flag=**true**;
29. **if**(!isEmpty()){
30. d=tos;
31. tos=tos->next;
32. }
33. **else** flag=false;
34. **return** flag;
35. }
 | 1. **bool** peek(Node\* d){
2. **bool** flag=**true**;
3. **if**(!isEmpty()){
4. d=tos;
5. }
6. **else** flag=**false**;
7. **return** flag;
8. }
9. **private**:
10. Node <T> \*tos;
11. };
12. **void** main (){
13. **int** a=10;
14. **int** b=20;
15. cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;
16. Stack <**int**> s;
17. Node <**int**> \*t;
18. t=**new** Node;

**56** t->data=a;**57**  t->next=NULL;**58** s.push(t);**59** t->data=b;**60** s.push(t);**61** s.pop(t);**62** a=t->data;**63** s.pop(t);**64** b=t->data;**65** cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;**66** } |

**3.2.4 Adapter design pattern.** The stack can be implemented using already existing linked list class through adapter design patterns ‎[12]. The two operations of linked list class i) *addonHead* and ii) *deletefromHead* are behavior wise same as *push* and *pop* operations of Stack class. This type of interface difference between two classes can be accommodated through the adapter design pattern. Adapter pattern lets another class work together with incompatible interfaces. Interfaces may be incompatible, but the inner functionality of both classes should complete the need. It is applicable when interface of an existing class does not match with actual requirements. Adapter pattern, in turn, can be implemented in two ways, by two different concepts of object oriented programming namely, inheritance and composition.

*3.2.4.1 Adapter pattern using inheritance:*The concept of inheritance can be used to implement the adapter pattern. Subclass (Stack) privately inherits the base class (LinkedList) and redefines new interfaces (*push*, *pop*) over base class interfaces (*addonHead*, *deletefromHead*). The Stack class can only call its defined functions i.e. push and pop. The Stack class uses already implemented logic in LinkedList class as a major advantage of reusability. Figure 6 shows a UML diagram of adapter pattern using inheritance. Here, code listing shows an adapter pattern through inheritance in which Stack class privately inheriting the methods (*addonHead*, *deletefromHead*) of the LinkedList class (Line 44-51).



Figure 6: UML diagram of implementing stack using inheritance with LinkedList

|  |
| --- |
| **Code Listing 7 : Adapter through inheritance in C++** |
| **1** **class** Node {**2** **public** :**3** **int** data;**4** Node \*next;**5** };**6** **class** LinkedList{**7** **public :****8** LinkedList( ) {**9** head=tail=NULL;**10** size=0;**11** }**12** **bool** addonHead(**int** d){**13** **if** (head==NULL){**14** head=**new** Node;**15** head->next=NULL;**16** }**17** **else**{ //non-empty-linkedList**18** Node \*temp=**new** Node**19** temp->data=d;**20** temp->next=head;**21** head=temp;**22** } **23** }**24** **bool** deleteFromHead(**int** &p){**25** **if** (head==NULL) //case 1**26** **return** **false**;**27** **else if**(head->next==NULL){//case 2 p= head->data; **28** **delete** head;**29** head=tail=NULL;}**30** **else**{ //case 3**31** Node \*temp=head; **32** p=head->data; | **33** head=head->next;**34** **delete** temp;**35** } **36** size=size-1;**37** **return** true;**38** }**39** **private:****40** Node \*head;**41** Node \*tail;**42** **int** size;**43** };**44** **class** Stack :**private** LinkedList{**45** **public** :**46** **bool** push(**int** p ) { **47** **return** addonHead(p);**48** }**49** **bool** pop(**int** & p) { **50** **return** deleteFromHead(p );**51** }**52** };**53** **void** main (){**54** **int** a=10;**55** **int** b=20;**56** cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;**57** Stack s;**58** s.push(a);**59** s.push(b);**60** s.pop(a);**61** s.pop(b);**62** cout<<"after Swapping a="<<a<<" and b="<<b<<endl;**63** } |

**2.2.2.2 Adapter pattern using composition.** The concept of composition can be used to implement the adapter pattern. This method is better than inheritance because of strong association between two classes ‎[4]. Most of the object oriented programming language support composition as compared to private inheritance. For example JAVA support composition but does not support private inheritance ‎[19]. Figure 7 shows a UML diagram of the adapter pattern using composition. Here code listing shows an adapter pattern through composition (Line 45-55).



Figure 7: UML diagram for implementation of Stack using composition of LinkedList

|  |
| --- |
| **Code Listing 8 : Adapter through composition in C++** |
| **1** **class** Node {**2** **public** :**3** **int** data;**4** Node \*next;**5** };**6** **class** LinkedList{**7** **public** :**8** LinkedList( ) {**9** head=tail=NULL;**10** size=0;**11** }**12** **bool** addonHead(**int** d){**13** **if** (head==NULL){ **14** head=**new** Node;**15** head->next=NULL;**16** }**17** **else**{//non empty linkedList**18** Node \*temp=**new** Node**19** temp->data=d;**20** temp->next=head;**21** head=temp;**22** } **23** }**24** **bool** deleteFromHead(**int** &p){**25** **if** (head==NULL) //case 1**26** **return false**;**27** **else if**(head->next==NULL){**28** p= head->data; //case2**29** **delete** head;**30** head=tail=NULL;}**31** **else** { //case 3**32** Node \*temp=head; **33** p=head->data;**34** head=head->next;**35** **delete** temp;**36** } | **37** size=size-1;**38** **return** true;**39** }**40** **private:****41** Node \*head;**42** Node \*tail;**43** **int** size;**44** };**45** **class** Stack {**46** **public** :**47** **bool** push(**int** p ) { **48** **return** list.addonHead(p);**49** }**50** **bool** pop(**int** & p) { **51** **return** list.deleteFromHead( p);**52** }**53** **private:****54** LinkedList list;**55** };**56** **void** main (){**57** **int** a=10;**58** **int** b=20;**59** cout<<"Before Swapping a="<<a<<" and b= "<<b<<endl;**60** Stack s;**61** s.push(a);**62** s.push(b);**63** s.pop(a);**64** s.pop(b);**65** cout<<"after Swapping a="<<a<<" and b= "<<b<<endl;**66** } |

**4**. **Relative Importance of the Implementation Variants**

In this section we highlight the relative importance of each variant so as to cover this topic of stack in an introductory course of data structures. We can clearly see from Figure 1 that a stack can be implemented using structured as well as object oriented programming techniques. There are two widely used methods for teaching computer programming courses, firstly object first approach is used; whereas, object later approach is the other variant. In the first case, the students learn the course of data structures after learning basic computer programming constructs, therefore, structured programming is used in this case. However, in the second case, the students have learned object oriented programming before studying the course of data structures. Therefore, in this case, object oriented paradigm is used to teach the course of data structures. Table 3 shows the importance of each implementation variants in terms of teaching the topic of Stack in a course of data structures. A brief description has been presented as remarks in the last column of the table to reason a certain recommendation level. Clearly, core means that the variant must be covered, recommended means that it should be covered subject to the provision of time, whereas, optional means that it may or may not be covered. In case of not covering a recommended or optional variants, we strongly recommend that these be considered in laboratory work, in assignments, or as supporting reading material of the topic.

**Table 3: Recommendation Level of Each Variant**

|  |  |  |
| --- | --- | --- |
| **Implementation Variant** | **Recommendation Level** | **Remarks** |
| Structured Programming | Core | Must be taught in case of imperative first approach of teaching computer programming course. Both Array and Linked List based implementations should be taught. |
| Object Oriented Programming | Core | Must be taught in case of object first approach or in case, object oriented programming has been taught prior to teaching data structures.Both Array and Linked List based implementations should be taught. |
| Generic Approach | Recommended | Subject to the provision of time it is recommended to teach. |
| Adapter Approach | Recommended | Subject to the provision of time it is recommended to teach. |
| Design Pattern Approach | Optional | May or may not be taught, but can be provided as additional resource. |

**5. Conclusion and Further work**

In this article, we define a taxonomy of stack from the perspective of its implementation using structured, and object oriented programming paradigms, and have also discussed its implementation using design patterns. We have explained all the variants of implementing stack using C++ programming language. We strongly believe that this taxonomy of implementing stack in different possible ways along with relevant details would help defining a better content list for different variants of the course of Data Structure.

**REFERENCES**

1. Goodrich, M., Tamassia, R., & Mount, D. (2007). DATA STRUCTURES AND ALOGORITHMS IN C++. John Wiley & Sons.
2. Drozdek, A. (2012). Data Structures and algorithms in C++. Cengage Learning.
3. Dastidar, G. D., Chattopadhyay, M., Chattopadhyay, S., & Ghosh, D. D. (2003). Data Structures through C Language.
4. Malik, D. S. (2010). C++ programming: Program design including data structures. Cengage Learning.
5. Stack (abstract data type), “http://en.wikipedia.org/wiki/Stack\_(abstract\_data\_type)#cite\_note-4” retrieved 19 November, 2014.
6. Sahami, M., Roach, S., Cuadros-Vargas, E., & Reed, D. (2012). Computer science curriculum 2013: reviewing the strawman report from the ACM/IEEE-CS task force. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 3-4). ACM.
7. Roberts, E., Shackelford, R., LeBlanc, R., & Denning, P. J. (1999). Curriculum 2001: Interim report from the ACM/IEEE-CS task force. In ACM SIGCSE Bulletin (Vol. 31, No. 1, pp. 343-344). ACM.
8. Cassel, L., Clements, A., Davies, G., Guzdial, M., McCauley, R., McGettrick, A, & Weide, B. W. (2008). Computer science curriculum 2008: An interim revision of CS 2001.
9. Gelfand, N., Goodrich, M. T., & Tamassia, R. (1998). Teaching data structure design patterns. In ACM SIGCSE Bulletin (Vol. 30, No. 1, pp. 331-335). ACM.
10. Hopcroft, J. E. (1983). Data structures and algorithms. Pearson education.
11. Niculescu, V. (2012). A DESIGN PATTERNS PERSPECTIVE ON DATA STRUCTURES. Acta Universitatis Apulensis, (30), 335-354.
12. Nguyen, D. (1998). Design patterns for data structures. In ACM SIGCSE Bulletin (Vol. 30, No. 1, pp. 336-340). ACM.
13. Ernst, D. J., Stevenson, D. E., & Wagner, P. J. (2009). Hybrid and custom data structures: evolution of the data structures course. ACM SIGCSE Bulletin, 41(3), 213-217.
14. Roberts, E., Shackelford, R., LeBlanc, R., & Denning, P. J. (1999). Curriculum 2001: Interim report from the ACM/IEEE-CS task force. In ACM SIGCSE Bulletin (Vol. 31, No. 1, pp. 343-344). ACM.
15. Cassel, L., Clements, A., Davies, G., Guzdial, M., McCauley, R., McGettrick, faWeide, B. W. (2008). Computer science curriculum 2008: An interim revision of CS 2001.
16. Sahami, M., Roach, S., Cuadros-Vargas, E., & Reed, D. (2012). Computer science curriculum 2013: reviewing the strawman report from the ACM/IEEE-CS task force. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 3-4). ACM.
17. Horowitz, Ellis: "Fundamentals of Data Structures in Pascal", page 67. Computer Science Press, 1984
18. C. L. Hamblin, "An Addressless Coding Scheme based on Mathematical Notation", N.S.W University of Technology, May 1957 (typescript).
19. Charatan, Q., & Kans, A. (2004). Formal software development: from VDM to Java. Palgrave Macmillan.
20. Muhammad Shoaib Farooq, Aqsa Ali, Adnan Abid (2014) What Should Be Taught About Arrays In Cs2? 2nd International Conference on Computational and Social Sciences (ICCSS-14), Vol (3), pp. 2904-2913.