Data Structures

Linear Structures

Pooja Singh

<pooja.singh@snu.edu.in>

Assistant Professor
Department of Computer Science
and Engineering
School of Engineering

January 2025

From the Previous Lecture

- **Programming languages provide simple data representations and operations on them**
- **For developing various applications, we may need to build our own data structures**
- **Abstract Data Types — encapsulation, mathematical abstractions, modularity, reusability**
- **Abstractions for collections of data — Sequences, Trees, Graphs**
- **Structures in modeling**
- **Structures in analysis and design**
- **Efficient representation and implementation of operations is a must!**

From the Previous Lecture

- **Programming languages provide simple data representations and operations on them**
- **For developing various applications, we may need to build our own data structures**
- **Abstract Data Types — encapsulation, mathematical abstractions, modularity, reusability**
- **Abstractions for collections of data — Sequences, Trees, Graphs**
- **Structures in modeling**
- **Structures in analysis and design**
- **Efficient representation and implementation of operations is a must!**
- **Any Questions?**

Shiv Nadar University

Sequences / Lists

- **Sequences are finite collection of objects of same type**
  - *(***1***, ***5***, ***9***)*
  - *(InsertCard , EnterPin, EnterAmount, CollectMoney , TakeOutCard)*

Shiv Nadar University

Sequences / Lists

- **Sequences are finite collection of objects of same type**
  - *(***1***,* **5***,* **9***)*
  - *(InsertCard , EnterPin, EnterAmount, CollectMoney , TakeOutCard)*
- **Sequence can be empty:** *()*

Shiv Nadar University

Sequences / Lists

- **Sequences are finite collection of objects of same type**
  - *(***1***,* **5***,* **9***)*
  - *(InsertCard , EnterPin, EnterAmount, CollectMoney , TakeOutCard)*
- **Sequence can be empty:** *()*
- **Like arrays, sequences are ordered collections and the order is linear — each object has only one previous and one next object**

Sequences / Lists

- **Sequences are finite collection of objects of same type**
  - *(***1***,* ***5****,* ***9****)*
  - *(InsertCard , EnterPin, EnterAmount, CollectMoney , TakeOutCard)*
- **Sequence can be empty:** *()*
- **Like arrays, sequences are ordered collections and the order is linear — each object has only one previous and one next object**
- **Unlike arrays, sequences are dynamic — objects can be inserted and deleted**

Shiv Nadar University

Sequences / Lists

- **Sequences are finite collection of objects of same type**
  - *(***1***,* **5***,* **9***)*
  - *(InsertCard , EnterPin, EnterAmount, CollectMoney , TakeOutCard)*
- **Sequence can be empty:** *()*
- **Like arrays, sequences are ordered collections and the order is linear — each object has only one previous and one next object**
- **Unlike arrays, sequences are dynamic — objects can be inserted and deleted**
- **Due to its dynamic nature, memory is dynamically allocated when needed and freed when not in use**

Sequences / Lists

- **Sequences are finite collection of objects of same type**
  - *(***1***,* **5***,* **9***)*
  - *(InsertCard , EnterPin, EnterAmount, CollectMoney , TakeOutCard)*
- **Sequence can be empty:** *()*
- **Like arrays, sequences are ordered collections and the order is linear — each object has only one previous and one next object**
- **Unlike arrays, sequences are dynamic — objects can be inserted and deleted**
- **Due to its dynamic nature, memory is dynamically allocated when needed and freed when not in use**
- **Sequences can be decomposed into sub-sequences**
- **Can also be decomposed into an object and the rest of the sequence — for example,** *(***1***,* **5***,* **9***)* **can be decomposed into 1, and** *(***5***,* **9***)*

Shiv Nadar University

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**
- **For example, in the list $(a_1, a_2, \cdots, a_n)$, a_i is in the i^{th} position**

SHIV NADAR UNIVERSITY

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**
- **For example, in the list** $(a_1, a_2, \cdots, a_n)$**,** a_i **is in the** i^{th} **position**
- **Position need not be an integer!**

SHIV NADAR UNIVERSITY

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**
- **For example, in the list** $(a_1, a_2, \cdots, a_n)$**,** a_i **is in the** i^{th} **position**
- **Position need not be an integer!**
- **First position is referred to as the beginning position**

SHIV NADAR UNIVERSITY

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**
- **For example, in the list** $(a_1, a_2, \cdots, a_n)$**,** a_i **is in the** i^{th} **position**
- **Position need not be an integer!**
- **First position is referred to as the beginning position**
- **There is also a legal position AFTER** a_n**, referred to as the end position; So, a sequence of** n **objects has** $n + 1$ **legal positions**
- **Concept of end position is required since a new object can be inserted in that position**

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**
- **For example, in the list** $(a_1, a_2, \cdots, a_n)$, a_i **is in the** i^{th} **position**
- **Position need not be an integer!**
- **First position is referred to as the beginning position**
- **There is also a legal position AFTER** a_n**, referred to as the end position; So, a sequence of** n **objects has** $n + 1$ **legal positions**
- **Concept of end position is required since a new object can be inserted in that position**
- **The begin and end positions are same in an empty list**

Position

- **There is a concept of position (similar to array index) and each position has a next (except for the last) and a previous (except for the first) position**
- **For example, in the list** $(a_1, a_2, \cdots, a_n)$**,** a_i **is in the** i^{th} **position**
- **Position need not be an integer!**
- **First position is referred to as the beginning position**
- **There is also a legal position AFTER** a_n**, referred to as the end position; So, a sequence of** n **objects has** $n + 1$ **legal positions**
- **Concept of end position is required since a new object can be inserted in that position**
- **The begin and end positions are same in an empty list**
- **Sequences are usually traversed starting from the first position and then following next positions until the end position is reached**

Shiv Nadar University

List Operations

- **CreateList()** — **creates and returns a new List structure**
- **DisposeList(l)** — **Destroys the list l and releases memory used by l**
- **IsEmptyList(l)** — **returns "TRUE" if l is empty and returns "FALSE" otherwise**
- **MakeEmptyList(l)** — **removes all the objects in l and resets it to an empty list**

List Operations

- **Insert(x, p, l): Object** x **is inserted at position** p **of list** l
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, x, a_p, \cdots, a_n)$
- p **can be any valid position including the end position**

Shiv Nadar University

List Operations

- **Insert(x, p, l): Object** x **is inserted at position** p **of list** l
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, x, a_p, \cdots, a_n)$
- p **can be any valid position including the end position**
- **Delete(p, l): Object at position** p **is deleted**
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, a_{p+1}, \cdots, a_n)$
- p **can be any valid position EXCEPT the end position**

List Operations

- **Insert(x, p, l): Object x is inserted at position p of list l**
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, x, a_p, \cdots, a_n)$
- p **can be any valid position including the end position**
- **Delete(p, l): Object at position p is deleted**
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, a_{p+1}, \cdots, a_n)$
- p **can be any valid position EXCEPT the end position**
- **Retrieve(p, l): Return the object at position p. List l is NOT modified!**

List Operations

- **Insert(x, p, l): Object x is inserted at position p of list l**
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, x, a_p, \cdots, a_n)$
- p **can be any valid position including the end position**
- **Delete(p, l): Object at position p is deleted**
- $(a_1, a_2, \cdots, a_{p-1}, a_p, \cdots, a_n)$ **—>** $(a_1, a_2, \cdots, a_{p-1}, a_{p+1}, \cdots, a_n)$
- p **can be any valid position EXCEPT the end position**
- **Retrieve(p, l): Return the object at position p. List l is NOT modified!**
- **Find(x, l): Returns the position of first occurrence of object x. Returns the end position if x is not found.**

Shiv Nadar University

Position Operations

- **Begin(l):** **Returns the begin position of list *l***
- **End(l):** **Returns the end position of list *l***
- **Next(p, l):** **Returns p's next position; not defined for end position**
- **Previous(p, l):** **Returns p's previous position; not defined for the beginning position**

Using List ADT

- **List ADT defines only the basic operations that need to know how a list is represented**
- **Other operations on list or applications using list can be implemented using these basic functions**

Using List ADT

- **List ADT defines only the basic operations that need to know how a list is represented**
- **Other operations on list or applications using list can be implemented using these basic functions**
- **For example, let us think of a function to remove the duplicate entries in a given list**
- **Given $(2, 3, 3, 2, 5, 1, 6, 1, 5)$, the function should return $(2, 3, 5, 1, 6)$**
- **How can we implement this using only the basic operations of List ADT?**

Shiv Nadar University

Purge List

```
void PurgeList(List l)
{
   Position p = Begin (l);

   while ( p != End (l) ) {
      Position q = Next(p);
      while ( q != End ( l ) ) {
         if ( Retrieve(p, l) == Retrieve(q, l) ) {
            Delete(q, l);
         }
         else q = Next ( q ) ;
      } // End of inner while loop
      p = Next ( p ) ;
   } // End of outer while loop
   return ;
} // End of PurgeList function
```

Shiv Nadar University

Queue

- **Queue, as the name suggests, is a special kind of a list where insertions happen only at the end and deletions happen only at the front**
- **It has only two recognized positions namely "front" and "last"**
- **It is also referred to as First-in-First-out (FIFO) structure**

Queue Operations

- **CreateQueue()** — **creates and returns a new Queue structure**
- **DisposeQueue(q)** — **Destroys the queue q and releases memory used by q**
- **IsEmptyQueue(q)** — **returns "TRUE" if q is empty and returns "FALSE" otherwise**
- **MakeEmptyQueue(q)** — **removes all the objects in q and resets it to an empty queue**

Shiv Nadar University

Enqueue

- **Enqueue(x, q) — add the object x to the Queue q**
  - $() \longrightarrow (x)$
  - $(e_1, e_2, \cdots, e_n) \longrightarrow (e_1, e_2, \cdots, e_n, x)$

- **Enqueue(x, q) — add the object x to the Queue q**
  - () —> (x)
  - $(e_1, e_2, \cdots, e_n)$ —> $(e_1, e_2, \cdots, e_n, x)$
- **There may be a capacity limit for a queue and in that case, enqueue may not always succeed**

SHIV NADAR UNIVERSITY

Front and Dequeue

- **Front(q) — returns the first object in the queue q (q is NOT modified!)**
  - $(e_1, e_2, \cdots, e_n)$ **—>** e_1
  - $()$ **—> ???**

Front and Dequeue

- **Front(q) — returns the first object in the queue q (q is NOT modified!)**
  - $(e_1, e_2, \cdots, e_n) \longrightarrow e_1$
  - $() \longrightarrow$ **???**
- **Dequeue(q) — removes the first object from the queue q**
  - $(e_1, e_2, \cdots, e_n) \longrightarrow (e_2, \cdots, e_n)$
  - $(x) \longrightarrow ()$
  - $() \longrightarrow$ **???**

Front and Dequeue

- **Front(q) — returns the first object in the queue q (q is NOT modified!)**
  - $(e_1, e_2, \cdots, e_n)$ **—>** e_1
  - $()$ **—> ???**
- **Dequeue(q) — removes the first object from the queue q**
  - $(e_1, e_2, \cdots, e_n)$ **—>** $(e_2, \cdots, e_n)$
  - (x) **—>** $()$
  - $()$ **—> ???**
- **It is also possible to combine both Front and Dequeue into a single operation**
  - **Dequeue(q) — removes and returns the first object from the q**

Queue as a wrapper around List

- **Queue may be easily implemented as a wrapper around a List**

Queue as a wrapper around List

- **Queue may be easily implemented as a wrapper around a List**
- **Enqueue(x,q) —> Insert(x, End(lst), lst)**

Queue as a wrapper around List

- **Queue may be easily implemented as a wrapper around a List**
- **Enqueue(x,q) —> Insert(x, End(lst), lst)**
- **Front(q) —> Retrieve(Begin(lst), lst)**

Queue as a wrapper around List

- **Queue may be easily implemented as a wrapper around a List**
- **Enqueue(x,q) —> Insert(x, End(lst), lst)**
- **Front(q) —> Retrieve(Begin(lst), lst)**
- **Dequeue(q) —> Delete(Begin(lst), lst)**

Queue as a wrapper around List

- **Queue may be easily implemented as a wrapper around a List**
- **Enqueue(x,q) —> Insert(x, End(lst), lst)**
- **Front(q) —> Retrieve(Begin(lst), lst)**
- **Dequeue(q) —> Delete(Begin(lst), lst)**
- **However, we will later learn about direct implementation of Queue ADT**

Using Queue ADT

- **Let us take a small example to use Queue ADT**
- **Write a function that takes a number and returns the reverse of that number**
- **For example, if 3792 is given, the function should return 2973**

Using Queue ADT

- **Let us take a small example to use Queue ADT**
- **Write a function that takes a number and returns the reverse of that number**
- **For example, if 3792 is given, the function should return 2973**
- **Split 3792 into 379 & 2, and Enqueue(2, q) —> *(*2*)***

Shiv Nadar University

Using Queue ADT

- **Let us take a small example to use Queue ADT**
- **Write a function that takes a number and returns the reverse of that number**
- **For example, if 3792 is given, the function should return 2973**
- **Split 3792 into 379 & 2, and Enqueue(2, q) —> *(*2*)***
- **Repeat this until all digits are processed resulting in *(*2*,* 9*,* 7*,* 3*)***

Shiv Nadar University

Using Queue ADT

- **Let us take a small example to use Queue ADT**
- **Write a function that takes a number and returns the reverse of that number**
- **For example, if 3792 is given, the function should return 2973**
- **Split 3792 into 379 & 2, and Enqueue(2, q) —> (2)**
- **Repeat this until all digits are processed resulting in (2, 9, 7, 3)**
- **Now, Dequeue digits one by one and construct the number 2973**

```
int Reverse_Num(int num)
{
  Queue q = CreateQueue();    // Constructor

  do {
    int tmp = num % 10; // Get digit at the unit place
    Enqueue(tmp, q);    // put it in the queue
    num = num / 10;     // Get the remaining number
  } while ( num != 0 ) ;
```

Shiv Nadar University

```
int rev = 0;

do {
    int tmp = Front(q);
    Dequeue(q);                 // Get the next digit
    rev = (rev * 10) + tmp;     // construct the number
} while (!IsEmpty(q));

DisposeQueue(q);

return rev;
} // End of Reverse_Num function
```

Stack

- **As the name suggests, Stack is a linear collection where objects are "stacked" one above the other**
- **Insertion and deletion can only happen at the "top"**
- **It is a special kind of a list where insertion and deletion happen at one end only**
- **There is only one recognized position called "top"**
- **It is also referred to as Last-in-First-out (LIFO) structure**

Stack Operations

- **CreateStack()** — **creates and returns a new Stack structure**
- **DisposeStack(s)** — **Destroys the stack s and releases memory used by it**
- **IsEmptyStack(s)** — **returns “TRUE” if s is empty and returns “FALSE” otherwise**
- **MakeEmptyStack(s)** — **removes all the objects in s and resets it to an empty stack**

Core Stack Operations

Push(x, s) — inserts object x at the top of the stack

- $() \longrightarrow (x)$
- $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow (x, e_n, e_{n-1}, \cdots, e_2, e_1)$

Shiv Nadar University

Core Stack Operations

- **Push(x, s) — inserts object x at the top of the stack**
  - $() \longrightarrow (x)$
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow (x, e_n, e_{n-1}, \cdots, e_2, e_1)$
- **Top(s) — returns the object at the top of the stack s (s is not modified!)**
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow e_n$
  - $() \longrightarrow$ **???**

Core Stack Operations

- **Push(x, s) — inserts object x at the top of the stack**
  - $() \longrightarrow (x)$
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow (x, e_n, e_{n-1}, \cdots, e_2, e_1)$
- **Top(s) — returns the object at the top of the stack s (s is not modified!)**
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow e_n$
  - $() \longrightarrow$ **???**
- **Pop(s) — removes the object at the top of the stack s**
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow (e_{n-1}, e_{n-2}, \cdots, e_2, e_1)$
  - $(x) \longrightarrow ()$
  - $() \longrightarrow$ **???**

Core Stack Operations

- **Push(x, s) — inserts object x at the top of the stack**
  - $() \longrightarrow (x)$
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow (x, e_n, e_{n-1}, \cdots, e_2, e_1)$
- **Top(s) — returns the object at the top of the stack s (s is not modified!)**
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow e_n$
  - $() \longrightarrow$ **???**
- **Pop(s) — removes the object at the top of the stack s**
  - $(e_n, e_{n-1}, \cdots, e_2, e_1) \longrightarrow (e_{n-1}, e_{n-2}, \cdots, e_2, e_1)$
  - $(x) \longrightarrow ()$
  - $() \longrightarrow$ **???**
- **Like in the case of a queue, Top(s) and Pop(s) may be combined into a single operation**

Stack as a wrapper around List

- **Like Queue ADT, Stack may be easily implemented as a wrapper around a List**
- **Push(x, s) —> Insert(x, Begin(lst), lst)**
- **Top(s) —> Retrieve(Begin(lst), lst)**
- **Pop(s) —> Delete(Begin(lst), lst)**
- **However, we will later learn about direct implementation of Stack ADT**

Using Stack ADT

- **Let us consider a small example of checking if a number is palindrome or not**

Using Stack ADT

- **Let us consider a small example of checking if a number is palindrome or not**
- **As in the case of the previous example, we will extract the digits one by one starting from the unit place**
- **Each digit will be both enqueued into a queue and pushed into a stack in the order of extraction**

Using Stack ADT

- **Let us consider a small example of checking if a number is palindrome or not**
- **As in the case of the previous example, we will extract the digits one by one starting from the unit place**
- **Each digit will be both enqueued into a queue and pushed into a stack in the order of extraction**
- **For example, given the number 47693, after extraction**
  - **Queue will be $(3, 9, 6, 7, 4)$**
  - **Stack will be $(4, 7, 6, 9, 3)$**

Using Stack ADT

- **Let us consider a small example of checking if a number is palindrome or not**
- **As in the case of the previous example, we will extract the digits one by one starting from the unit place**
- **Each digit will be both enqueued into a queue and pushed into a stack in the order of extraction**
- **For example, given the number 47693, after extraction**
  - **Queue will be $(3, 9, 6, 7, 4)$**
  - **Stack will be $(4, 7, 6, 9, 3)$**
- **Now compare Front(q) and Top(s); If they are same, then Dequeue(q) and Pop(s); Repeat this until they become empty**
- **If any mismatch is found then the given number is not a palindrome; otherwise, it is a palindrome**

Shiv Nadar University

Palindrome

```
BOOL IsPalindrome(long num)
{
  Queue q = CreateQueue(); // Constructor for queue
  Stack s = CreateStack(); // Constructor for stack

  do {
    int tmp = num % 10;
    Enqueue(tmp, q);
    Push(tmp, s);
    num = num / 10;
  } while (num != 0);
```

Shiv Nadar University

```
while (!IsEmptyStack(s) && !IsEmptyQueue(q)) {
   if ( Front(q) != Top(s) ) {
      DisposeQueue(q);
      DisposeStack(s);
      return FALSE ;
   }
   Dequeue(q);
   Pop(s);
}

DisposeQueue(q);
DisposeStack(s);

return TRUE;
} // End of IsPalindrome function
```

Summary

- We have discussed List abstractions and the associated Position concept
- We have explored various basic operations on List and Position
- We have seen some code to work with the definition of List interface
- Queue is a FIFO abstraction that could be implemented as a wrapper around a List (we will see independent implementation of Queue later)
- Similarly, Stack which is a LIFO abstraction could also be implemented as a wrapper around List
- We have discussed a couple of simple applications using Queue and Stack ADTs

What next?

- **We will focus on algorithm analysis in the next couple of lectures**
- **We will then explore different implementations of List ADT**
- **We will look at some of the applications of Lists — implementing polynomials and radix sort**
- **Later we will discuss implementations of Stack ADT and Queue ADT**