Chapter 2 – List ADT

Abstract Data Types

Abstract Data Types are Data structures that hide the details of the implementation.

- ADTs provide an interface methods to the users so that they can use the data structure but they need not know how it is implemented.

- ADT implementation also deals with efficiency – it ensures that operations (methods) are optimally implemented

- ADTs in Java are written to accept generic types of data.

Arrays as ADTs

So far you have been dealing with arrays of specific type of data. Such as array of int, array of float, array of String or array of any objects.
Let us see how to create an Array ADT that contains array of any generic object and also provide methods for common things that are done with an array.

We need to decide the interface methods that we provide to the user.

With most data structures (ADT), we will provide one or more constructors, toString() method, empty() and size() methods. Ability to add and remove elements from the data structure and any other methods that are applicable to that data structure are also provided.

Arrays as ADTs

For arrays we would like to provide the following methods:

```
public ArrayADT()   // creates an array
public ArrayADT(int capacity) // creates an array of given capacity
public int size() // returns the size
public boolean empty() // returns if array is empty
public void add( T element) // adds element to the end of the array
public void addAll(ArrayADT<T> array) // adds all elements
public String toString() // converts the array to string
public int search(T target) // returns the index of target
public T getElement(int index) // returns element at index
public void setElement(T elem, int index) // sets the item at index
to elem.
```


Arrays as ADTs

Let us how to implement the Array ADT class:
Note that all variables within a data structure are kept private.

```
public class ArrayADT<T>{

   private final int DEFAULT_CAPACITY = 100;
   private final int NOT_FOUND = -1;

   private int count;
   private T[] contents;

  public ArrayADT(){
    count = 0;
    contents = (T[]) (new Object[ DEFAULT_CAPACITY]);
  }
```


Arrays as ADTs

```
public ArrayADT(int capacity){
    count = 0;
    contents = (T[]) (new Object[capacity]);
  }

public int size(){
    return count;
  }

  public boolean empty(){
    return (count == 0);
  }
```

Arrays as ADTs

```
public void add( T element){
      if (size() == contents.length)
        extendCapacity();
      contents[count] = element;
      count++;
 }

 private void extendCapacity(){
      T[] larger = (T[]) (new Object[contents.length*2]);
      for (int index = 0; index < contents.length; index++)
          larger[index] = contents[index];
      contents = larger;
}
```


Exceptions from ADT methods

Note that we must throw exception (from either the well known exception or make our own if needed) when the incoming data is not correct.

```
public T getElement(int index) throws
IndexOutOfBoundsException{
   if (  index <0 || index >= size())
             (throw (new IndexOutOfBoundsException());
   else
             return(contents[index]);
}
```


Iterator of ADTs

To travel through a list data data structure we would like to implement an iterator method in the ArrayADT class.

In order to obtain an iterator that iterates over the array let us make an ArrayIterator class that implements the Iterator interface. Implementing an interface requires us to implement the abstract methods in that interface.

ArrayIterator Class

To implement an Iterator interface, we must implement the abstract methods hasNext(), next() and remove()

```
import java.util.*;
public class ArrayIterator<T> implements Iterator<T>{

private int count;
private int current;
private T[] items;

public ArrayIterator(T[] contents, int size){
   items = contents;
   count = size;
   current = 0;
}
```


ArrayIterator Class

```
 public boolean hasNext(){
      return (current < count);
}

 public T next(){
      current++;
      return items[current-1];
}

  public void remove() throws UnsupportedOperationException{
       throw new UnsupportedOperationException();
  }
}
```


ArrayIterator Class

Now we can add the following method to the ArrayADT class.

```
public Iterator<T> iterator(){
     return new ArrayIterator<T> (contents,count);
  }
public String toString(){

        String rString = "";
        Iterator<T> scan = iterator();
         while (scan.hasNext())
           rString = rString + (scan.next()).toString();
        return(rString);

}
```

You may now complete the data structure by writing all other methods in the ArrayADT

Using ArrayADT class

```
public class MainClass{
    public static void main(String[] args){
        ArrayADT<String> example = new ArrayADT<String>();
         example.add("Apples");
         example.add("Oranges");
         System.out.println(example.toString());
        if ( !example.search(“Peaches”))
             example.add(“Peaches”);
        System.out.println(example.get(2));
        Iterator eI = example.iterator();
        while (eI.hasNext()){
         System.out.println(eI.next());
    }
```


Vector class in java.util

It is a list data structure that helps add objects and remove objects from the list. Here are some useful methods and constructors.

<u>Constructors:</u>

Vector() – makes an empty vector

Vector(initCap) – makes a vector with the initial capacity given by initCap

<u>Methods:</u>

boolean add (obj) – adds object to the end of the vector

boolean add(pos, obj), adds the object at pos after moving the elements to the right by one position,

Vector class in java.util

<u>Methods:</u>

int capacity() – returns the capacity of the vector

Object element(pos) – returns the Object at a given position.

boolean remove(obj) – removes object if it is there.

Object remove(pos) – removes the object at position pos.

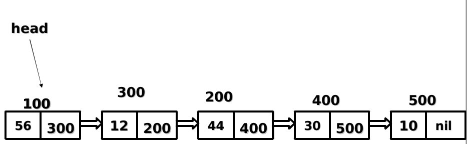
int indexOf(obj) – returns the index of obj if it is there.

int indexOf(obj, pos) – returns the index of the obj, searching starting at position pos

2.2 Linked Lists

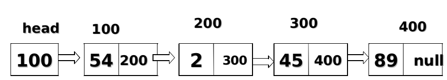
2.2 Linked Lists

- Data is not stored contiguously like arrays, but they are linked like a chain. You have to start at the beginning and work your way through to the end, in order to traverse the list.
- You can only traverse one way in a singly linked lists!



2.2 Linked Lists (cont)

```
class LinearNode{
          int element ;
          LinearNode next;
}
```



2.2 Linked Lists

```
public class LinearNode<T>{
    private LinearNode<T> next;
    private T element;
    public LinearNode()
    public LinearNode(T elem)
    public LinearNode<T> getNext()
    public T getElement()
    public void setElement(T elem)
    public void setNext(LinearNode<T> node)
    public boolean equalElement(LinearNode<T> node)
        throws NullPointerException
}
```


2.2 Basic Operations on Linked Lists

```
public class LinkedList<T>{
    private LinearNode<T> head;
    private LinearNode<T> tail;
//Constructs an empty list
    public LinkedList()
// adds to the end of the List
    public void addLast(T elem)
//   returns an iterator for this class
    public LinkedListIterator iterator()
// add at the position index
    public void add(T elem, int index) throws IndexOutOfBoundsException
// adds to the beginning of the list
   public void addFirst(T elem)
// returns the index of first occurrence of elem if it is there.
   public int contains(T elem)
```


2.2 Basic Operations on Linked Lists

```
// returns the element at a given index
  public T  get(int index) throws IndexOutOfBoundsException
// returns the first element
    public T getFirst()
// returns the last element
     public T getLast()// removes the item at position index and returns it
      public T remove(int index)throws IndexOutOfBoundsException
// removes the first item and returns it
      public T removeFirst()
// removes the last item and returns it
      public T removeLast()
// searches for elem and removes the first occurrence of elem it and returns true/false
      public boolean remove(T elem)
// returns the number of elements in the list
      public int size()
// returns the string representation of the elements in the list
      public String toString()
```

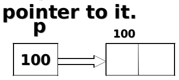

Linked List Iterator

```
public LinkedListIterator(LinearNode<T> head)
public boolean hasNext()
public T next()
public void remove() throws UnsupportedOperationException
```

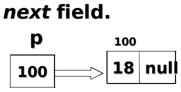

2.2 Constructor for LinearNode

Let us say we want to create a node with key= 18

Step 1: Make a node and get a pointer to it.



Step 2: Fill the *element* and the *next* field.



2.2 Constructor for LinearNode

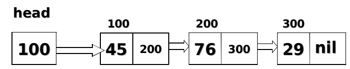
```
public LinearNode(T elem){
     this.element = elem;
     this.next = null;
}
```

Gets and Sets in LinearNode

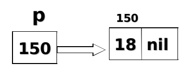
```
public T getElement(){

   return(this.element)
}
public void setElement(T elem){
   this.element = elem;
 }
public T getNext(){

   return(this.next)
}
public void setNext(LinearNode node){
   this.next = node;
}
```


2.2 addFirst

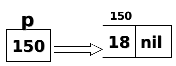
Given the list below, let us insert 18 into the list at the beginning



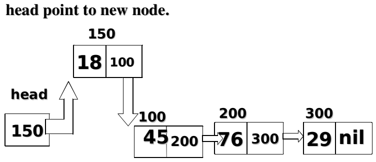
Step 1: Let us first make a new node with 18 in it.



2.2 addFirst



Step 2: Let us then make new node point to current head and head point to new node.

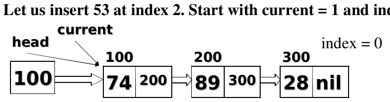


2.2 addFirst

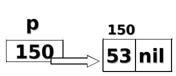
```
public void addFirst(T elem){
      LinearNode<T> e = new LinearNode(elem);
      if (head == null){ head = e; tail = e;}
      else
       e.setNext(head);
      head = e;
   }
```


2.2 add at an index

Let us insert 53 at index 2. Start with current = 1 and index = 0

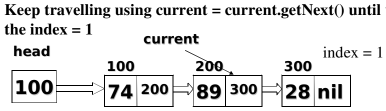


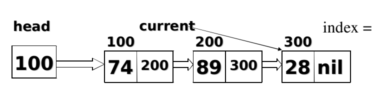
Step 1: Let us first make a new node with 53 in it.



2.2 add at an index

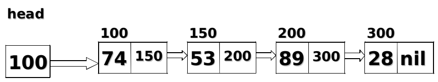
Keep travelling using current = current.getNext() until we reach the index = 1





2.2 add at an index

Step 2: Now make the new node point to 74's 'next' and 74 point to the new node.

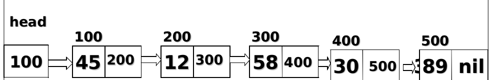


2.2 add

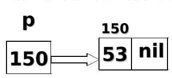
```
public void add(T elem, int index) throws IndexOutOfBoundsException{
    if (index < 0 || index > size())
          throw (new IndexOutOfBoundsException());
    else{
        if (index == 0) addFirst(elem);
        else if (index == size()) addLast(elem);
             else{
                  LinearNode<T> toInsert = new LinearNode(elem);
                  LinearNode<T> curr = head;
                  for (int count = 1; count < index; count++){
                      curr = curr.getNext();
                  }
                  toInsert.setNext(curr.getNext());
                  curr.setNext(toInsert);
             }
    }
```


2.2 Insert at the end

Let us insert 53 at the end of the list below.

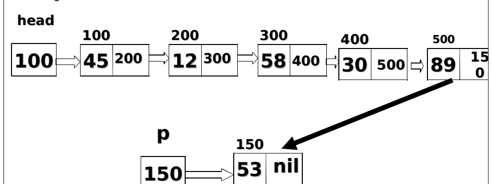


Step 1: Let us first make a new node with 53 in it.



2.2 Insert at the end

Step 2:Find the last node and set the *next* field of the last node to p.

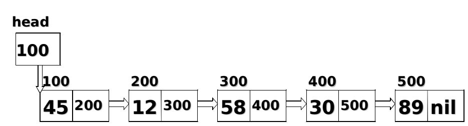


2.2 addLast

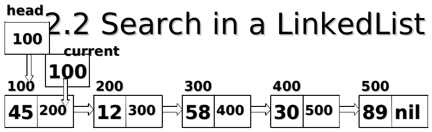
```
public void addLast(T elem){
      LinearNode<T> e = new LinearNode(elem);
      if (head == null) { head = e; tail = e;}
      else{
            tail.setNext(e);
            tail = e;
      }
   }
```


2.2 Search in a Linked List

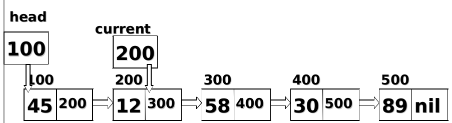
Search for key = 30 in the list below.



2.2 Search in a LinkedList

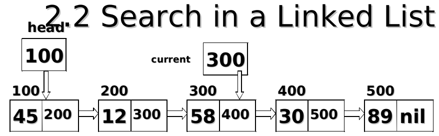


Compare 30 and *current.getElement()* which is *45* **found=false**

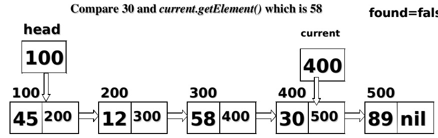


Compare 30 and *current.getElement()* which is *12* **found=false**

2.2 Search in a Linked List



Compare 30 and *current.getElement()* which is 58 **found=false**



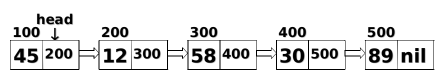
Compare 30 and *current.getEkement()*which is 30 STOP **found=true**

2.2 contains

```
public int contains(T elem) {
    LinkedListIterator<T> lt = iterator();
    int count = -1;
    boolean found = true;
    while (lt.hasNext() && !found){
        T currElem = lt.next();
        if (currElem.equals(elem)) found = true;
        count++;
    }
    if (!found)
       count = -1;
       return(count);
    }
```


2.2 Finding the previous node to a given index

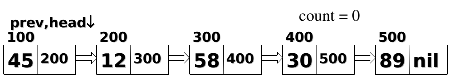
Given the index of the node that contains 400 as 3,, find the node that precedes it.



2.2 Finding the node at previous index

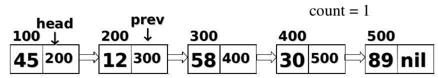
Start with *prev* as *head* and keep moving using prev = prev.getNext(). Start with counter of 0. Repeat this until the counter reaches 2

count = 0

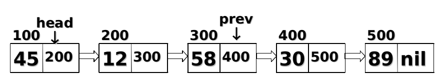


2.2 Find previous node to given node

count = 1



count = 2

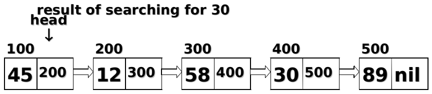


Since count = 2, STOP

2.2 Delete from Linked List

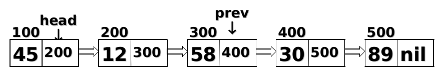
If we want to delete 30 from the list below, we first find the index of the node containing 30, using contains routine, then find the previous node to 30, using the *procedure we saw in previous slides. We then* make changes to pointers to delete the node containing 30.

Let us assume that index was 3 was the result of searching for 30

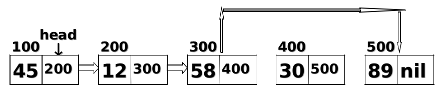


2.2 Delete From Linked List

Let us assume that *prev* was the result of finding previous to *30*

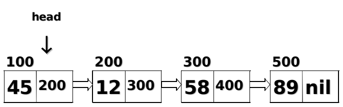


We make *prev.next* to equal *current.next.*



2.2 Delete From a Linked List

We get a linked list shown below, with the node containing 30 deleted from the list.



2.2 remove

```
public T remove(int index)throws IndexOutOfBoundsException{
        if (index < 0 || index > size())
            throw (new IndexOutOfBoundsException());
        else{
            T rValue = null;
            if (index != 0){
               LinearNode<T> prev= head, curr = head.getNext();
              for (int count = 1; count < index; count++){
                 prev = curr;
                curr = curr.getNext();
              }
              rValue = curr.getElement();
              prev.setNext(curr.getNext());
            }
            else rValue = removeFirst();
            return(rValue);
        }
    }
```


2.2 In Class Activity

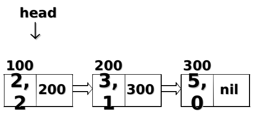
Write an algorithm to insert an element into sorted singly linked list so that the resulting list also contains elements in the sorted order.

2.3 – Application of Linked Lists

2.3 Polynomials as Linked Lists

Storing and manipulate polynomials can be done using singly linked lists.

For example, $2x^2+3x+5$ can be stored as a linked list containing (base, exponent) pairs as data part of the node.



2.3 In Class Activity

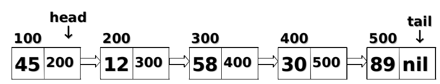
- Write an algorithm to add two polynomials, stored as linked lists in poly1 and poly2.

2.4 – Variations of Linked Lists

2.4 Singular Linked Lists with tail

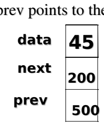
There are linked lists in which both *head* and *tail* are maintained

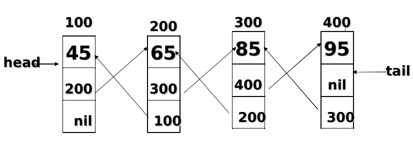
Note the the tail only points to the last element, it does not help in traversal since there is no way to go back to using tail.



2.4 Doubly Linked Lists

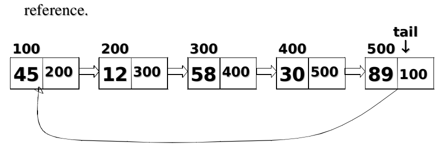
- In doubly linked list every node has three fields, data, next and prev. The prev points to the previous node.





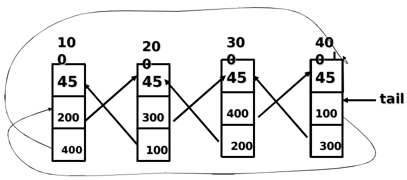
2.4 Circular Lists

- Circular lists are those in which the last node points back to the first node.
- In a circular singly list, only tail is used as a permanent reference.



2.4 Circular Lists

- In a circular doubly linked lists, again tail is the only permanent pointer used.



2.5 – Linked List Class in JDK

2.5 Linked Lists in JDK-java.util

Constructor Summary

LinkedList()
Constructs an empty list.

LinkedList(Collection c)
Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary

void **add**(int index, Object element)
Inserts the specified element at the specified position in this list.

2.5 Linked List in JDK- java.util

boolean **add**(Object o)
Appends the specified element to the end of this list.

boolean **addAll**(Collection c)
Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

boolean **addAll**(int index, Collection c)
Inserts all of the elements in the specified collection into this list, starting at the specified position.

void **addFirst**(Object o)
Inserts the given element at the beginning of this list.

void **addLast**(Object o)
Appends the given element to the end of this list.

2.5 Linked List in JDK-java.util

void **clear**()
Removes all of the elements from this list.

Object **clone**()
Returns a shallow copy of this LinkedList.

boolean **contains**(Object o)
Returns true if this list contains the specified element.

Object **get**(int index)
Returns the element at the specified position in this list.

Object **getFirst**()
Returns the first element in this list.

Object **getLast**()
Returns the last element in this list.

2.5 Linked List in JDK-java.util

int **indexOf**(Object o)
Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

int **lastIndexOf**(Object o)
Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

ListIterator **listIterator**(int index)
Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.

Object **remove**(int index)
Removes the element at the specified position in this list.

boolean **remove**(Object o)
Removes the first occurrence of the specified element in this list.

2.5 Linked List in JDK-java.util

Object **removeFirst**()
Removes and returns the first element from this list.

Object **removeLast**()
Removes and returns the last element from this list.

Object **set**(int index, Object element)

Replaces the element at the specified position in this list with the specified element.

int **size**()
Returns the number of elements in this list.

Object[] **toArray**()
Returns an array containing all of the elements in this list

Object[] **toArray**(Object[] a)
Returns an array containing all of the elements in this list.

2.6 Exercises

2.6 Exercises

1. Write an algorithm to find the node previous to the last node in a singly linked list with head only.
2. Develop all the operations we have covered for singly linked lists with head and tail. Do any of the time complexities change, if so explain.
3. Develop all the operations we have covered for doubly linked lists. Do any of the time complexities change, if so explain.
4. Develop all the operations we have covered for circular singly linked lists. Do any of the time complexities change, if so explain.
5. Develop all the operations we have covered for circular doubly linked lists. Do any of the time complexities change, if so explain.

2.6 Exercises

6. Specify Operations for a simple linked list that does the following:
   a. Makes the ith node in the list the current node.
   b. Makes the node whose element has a specified key value the current node.
   c. Sorts the list in ascending order based on the values of the key field of the elements.
7. Write procedures that implement the operations in Exercise above. These procedures should make use of the operations specified for the simple linked list covered in class.
8. Write an algorithm that implements each of the following operations for a singly linked list.
   a. Exchange the current node with its successor.
   b. Exchange the current node with its predecessor.

2.6 Exercises

9. Write an algorithm to reverse elements in a singly linked list by doing only one pass through the list. What is the time complexity of your algorithm. Demonstrate the algorithm on a list with five elements.
10. Write an algorithm to merge two linked list in sorted order into a resulting linked list that is also in sorted order. What is the time complexity of your algorithm.
11. How would you implement linked lists using arrays. What are the advantages and disadvantages of such an implementation.
12. Write an algorithm to check whether two singly linked lists have the same contents although in different order.
13. Write an algorithm to attach a singly linked list to the end of another singly linked list.

2.6 Exercises

14. Write an algorithm to count the number of nodes in a singly linked list.
15. How would you adapt binary search to singly linked lists. What would be the time complexity of the adapted algorithm.
16. Write an algorithm to retrieve the center element in a singly linked list.