GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 2 Report

Muhammed Bedir ULUCAY

1901042697

ArrayList Time Complexity:

```java
/**Adding new element to ArrayList*/
public boolean add(E anEntry) {
    if (size == capacity)
        reallocate();

    theData[size++] = anEntry;
    return true;
}
```

Amortized constant time Θ(1)

If array full then we grow up the size and it takes Θ(n)

```java
/**Insert the getting element wanted position*/
public void add(int index, E anEntry) {
    if (index < 0 || index > size)
        throw new ArrayIndexOutOfBoundsException(index);

    if (size == capacity)
        reallocate();

    // Shift data in elements from index to size - 1
    for (int i = size; i > index; i--)
        theData[i] = theData[i-1];
    // Insert the new item.
    theData[index++] = anEntry;
    size++;
}
```

Linear time Θ(n)

Because we can need to shift array ex add index = 0 we need to shift all array to index+1

```java
/**Return the wanted index element*/
public E get(int index) {
    if (index < 0 || index >= size)
        throw new ArrayIndexOutOfBoundsException(index);

    return theData[index];
}
```

Constant Time Θ(1)

Array has random access so it can directly acces the wanted index of itself

```java
/**Return the index of e where the same in ArrayList if not in arrayList return -1*/
public int indexOf(E e) {
    for(int i=0; i<size; ++i) {
        if(get(i).equals(e))
            return i;
    }

    return -1;
}
```

Linear Time Θ(n)

This method compare object 0 – size one by one it is making linear search

```java
/**Array Capacity increasing by size*2*/
private void reallocate() {
    capacity = 2 * capacity;
    theData = Arrays.copyOf(theData, capacity);
}
```

Linear Time Θ(n)

Arrays.copyOf() takes linear time because it assign all theData object tmp array then return

The copied array as size = size * 2

```java
/**Remove the element index in Array List and if its need to shift the list*/
public E remove(int index) {
    if (index < 0 || index >= size)
        throw new ArrayIndexOutOfBoundsException(index);

    E returnValue = theData[index];
    for (int i = index + 1; i < size; i++)
        theData[i - 1] = theData[i];

    size--;
    return returnValue;
}
```

Linear time Θ(n)

Because it can need to shift all array ex remove(0) all array obj must be shift as implemented

```java
/**Replace the element where we get as parameter in index */
public E set(int index, E newValue) {
    if (index < 0 || index >= size)
        throw new ArrayIndexOutOfBoundsException(index);

    E oldValue = theData[index];
    theData[index] = newValue;
    return oldValue;
}
```

Constant Time Θ(1)

Array has random acces so it can direclty acces the location of wanted index

```java
private class Iter implements Iterator<E>{

    int index;
    int lastRet;
    Iter(){
        index = 0;
        lastRet = -1;
    }

    @Override
    public boolean hasNext() {
        return index < size;
    }

    @Override
    public E next() {
        if(index >= size)
            throw new NoSuchElementException();
        lastRet++;
        return get(index++);
    }

    @Override
    public void remove() {
         if (lastRet < 0)
                throw new IllegalStateException();

         ArrayList.this.remove(lastRet);
    }

}

@Override
public Iterator<E> iterator() {
    return new Iter();
}
```

HasNext() -> Θ(1) it just an logic comparision

Next() -> Θ(1) it can acces the direct the wanted location in array

Remove() -> Θ(n) it can want to shift the array so it takes linear time

Iterator<E> -> Θ(1) it just make an object and return it dont have loop

Linked List Time Complexity:

```java
/** Append item to the end of the list
@param item The item to be appended
@return true (as specified by the Collection interface)
*/
public boolean add(E item) {
    add(size, item);
    return true;
}
```

Linear Time -> Θ(n)

Because in linked list if you want to add it end of the list you need to

came from head of list as we implement

```java
/** Insert the specified item at index
@param index The position where item is to be inserted
@param item The item to be inserted
@throws IndexOutOfBoundsException if index is out of range
*/
public void add(int index, E item) {
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException(Integer.toString(index));

    if (index == 0) {
        addFirst(item);
    } else {
        Node<E> node = getNode(index-1);
        addAfter(node, item);
    }
    size++;
}
```

Linear Time -> Θ(n)

It is also use getNode(int) and its complexity Θ(n) other operations Θ(1)

```java
/** Add a node after a given node
@param node The node preceding the new item
@param item The item to insert
*/
private void addAfter(Node<E> node, E item) {
    node.next = new Node<>(item, node.next);
}
```

Constant time -> Θ(1)

It just adding a new element wanted place we get node as paramater

Because of that dont need to iter all over we have it already

```java
/** Get the data at index
@param index The position of the data to return
@return The data at index
@throws IndexOutOfBoundsException if index is out of range
*/
public E get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException(Integer.toString(index));

    Node<E> node = getNode(index);
    return node.data;
}
```

Linear Time -> Θ(n)

GetNode(int) time complexity Θ(n) to iter head to tail each time

```java
/** Find the node at a specified position
@param index The position of the node sought
@return The node at index or null if it does not exist
*/
private Node<E> getNode(int index) {
    Node<E> node = head;
    for (int i = 0; i < index && node != null; i++)
        node = node.next;

    return node;
}
```

Linear Time -> Θ(n)

To reach the wanted place we need to start from head and iter the list

```java
/** Add an item to the front of the list.
@param item The item to be added
*/
private void addFirst(E item) {
    head = new Node<>(item, head);
}
```

Constant time -> Θ(1)

It just add new element head

```java
/** Returns the index of the first occurrence of the specified element
 * in this list, or -1 if this list does not contain the element.
 * @param o element to search for*/
public int indexOf(E data) {
    Iterator<E> it = this.iterator();
    for(int i=0; it.hasNext() ; ++i)
        if(it.next().equals(data))
            return i;

    return -1;
}
```

Linear time -> Θ(n)

It using iterator and iter all list

```java
/** Remove the first occurrence of element item.
@param item The item to be removed
@return removed element if it is not exist return null.
*/
public E remove(E item) {
    if(head.getData().equals(item))
        return removeFirst();

    return removeAfter(getNode(indexOf(item)-1));
}
```

Linear time -> Θ(n)

It is using actually indexOf Θ(n) + getNode Θ(n) => Θ(2n)

Θ(2n) = is equal to Θ(n)

```java
private E removeAfter(Node<E> node) {
    Node<E> temp = node.next;
    if (temp != null) {
        node.next = temp.next;
        size--;
        return temp.data;
    }
    return null;
}
```

Constant time -> Θ(1)

It just doing constant operation over getting node parameter

```java
/** Remove the first node from the list
@return The removed node's data or null if the list is empty
*/
private E removeFirst() {
    Node<E> temp = head;
    if (head != null)
        head = head.next;
    // Return data at old head or null if list is empty
    if (temp != null) {
        size--;
        return temp.data;
    }
    return null;
}
```

Constant time -> Θ(1)

It just doning constant operations

```java
  /** Remove the first occurrence of element item.
  @param index to removed
  @return removed element if it is not exist return null.
  */
  public E remove(int index) {
      if (index < 0 || index >= size)
          throw new IndexOutOfBoundsException(Integer.toString(index));

      if(index == 0)
          return removeFirst();

      return removeAfter(getNode(index-1));
  }
```

Linear Time -> Θ(n)

It is using removeAfter - > it complexity Θ(n)

```java
/** Store a reference to anEntry in the element at position index.
@param index The position of the item to change
@param newValue The new data
@return The data previously at index
@throws IndexOutOfBoundsException if index is out of range
*/
public E set(int index, E newValue) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException(Integer.toString(index));

    Node<E> node = getNode(index);
    E result = node.data;
    node.data = newValue;
    return result;
}
```

Linear Time -> Θ(n)

It is using getNode -> Θ(n)

```java
private class Itr implements Iterator<E>{

    private Node<E> lastReturned;
    private Node<E> next;
    private int nextIndex;
    Itr(){

    @Override
    public boolean hasNext() {
        return nextIndex < size;
    }

    @Override
    public E next() {
         if (!hasNext())
               throw new NoSuchElementException();

         lastReturned = next;
         next = next.next;
         nextIndex++;
         return lastReturned.data;
    }

    @Override
    public void remove() {
        if (lastReturned == null)
            throw new IllegalStateException();

        Node<E> lastNext = lastReturned.next;

        removeAfter(getNode(nextIndex-2));
         if (next == lastReturned)
               next = lastNext;
         else
               nextIndex--;
         lastReturned = null;
    }
}
```

HasNext -> contant time -> Θ(1)

Next -> constant time -> Θ(1) because keeping the next referance

Remove -> Linear Time -> Θ(n) it is using getNode

```java
@Override
public Iterator<E> iterator() {
    return new Itr();
}
```

Constant time -> Θ(1)

Creating new obj and return it constant operations

Hybrid List Time Complexity:

We use size as n (Number of Furniture)

Number of Linked list node = n

MaxNumber array list size = m

```
/**Adding element end of the list*/
public boolean add(E e) {
    if(size == 0)
        addFirst(e);
    else
        addLast(e);
    elementNumber++;
    return true;
}
```

Time Complexity -> O(n)

There are two different option for this if we add the first element it is constant but if we add some furniture then call this we call addLast it is constant so O(n)

```
/**Adding the first element of HybridList create and LinkList and
 * create first element of LinkList it is ArrayList*/
private void addFirst(E e) {
    datas = new LinkedList<>();
    datas.add(new ArrayList<>());
    datas.get(0).add(e);
    size++;
}
```

Constant Time -> Θ(1)

Creating new object and assign

get from linked list get(0) -> constant Θ(1)

add from array list add(e) -> constant Θ(1)

```
/**Returning the wanted arrayList as node*/
public ArrayList<E> get(int index){
    return datas.get(index);
}
```

Linear time -> Θ(n)

It is using linked list get method => Θ(n)

```java
/**Adding new Element end of the Hybrid List*/
private void addLast(E e) {

    Iterator<ArrayList<E>> listIter = datas.iterator();

    while(listIter.hasNext()) {
        ArrayList<E> tmp = listIter.next();
        if(tmp.getSize() < MAX_NUMBER) {
            tmp.add(e);
            return;
        }
    }

    datas.add(new ArrayList<>());
    datas.get(size).add(e);
    size++;
}
```

Linear time -> Θ(n)

We iter over Array Node so if we have n furniture it will work n/10 at least that means time complexity Θ(n)

```java
/**Return the index of wanted elemnt in Hybrid list
 * If it not in List return -1*/
public int indexOf(E e) {
    if(size == 0) return -1;
    HIterator<E> HIter = this.iterator();
    int counter = 0;

    while(HIter.hasNext()) {
        Iterator<E> AIter = HIter.next().iterator();
        while(AIter.hasNext()) {
            if(AIter.next().equals(e))
                return counter;
            counter++;
        }
    }

    return -1;
}
```

Time Complexity -> Θ(n * m)

This functions is compare furniture one by one

```java
/**Remove index value in Hybrid List*/
public E remove(int index) {
    if(index > elementNumber)
        throw new IndexOutOfBoundsException(Integer.toString(index));

    HIterator<E> HIter = this.iterator();
    for(int i=0; i<index / MAX_NUMBER; ++i)
        HIter.next();       //To pass ArrayList (index / MAX_NUMBER) time

    Iterator<E> AIter = HIter.next().iterator();
    int nth = index - ((index / MAX_NUMBER) * 10);  // (index / MAX_NUMBER) 67/10 = 6 -> 67-(6*10) = 7
    for(int i=0;i<nth; ++i)                         // we want 7 small step
        AIter.next();       //To pass element using integer specification

    E tmp = AIter.next();
    AIter.remove();
    return tmp;
}
```

Time Complexity -> Θ((n / m) + m)

This function also use big and small step it is work big steps pass node
Small steps iter over arrayList its count max_number
its work totaly = (index / max_number) + max_number = (n / m) + m

```java
/**Iterator for Hybrid List*/
private class HIter implements HIterator<E>{

    Iterator<ArrayList<E>> arrIt;
    int index;

    HIter() {
        arrIt =  datas.iterator();
        index = 0;
    }

    public boolean hasNext() {
        return arrIt.hasNext();
    }

    public ArrayList<E> next() {
        ArrayList<E> tmp = arrIt.next();
        return tmp;
    }

    public Iterator<E> inIterator(){
        return datas.get(index).iterator();
    }

    public void remove() {
        datas.remove(datas.get(--index));
    }

}
```

HasNext() - > constant(1)

Next() -> constant(1)

InIterator() -> Θ(n) it is using linked list get method

Remove() -> Θ(n) it is using linked list get method and remove method Θ(2n) = Θ(n)

```java
@Override
public HIterator<E> iterator() {
    return new HIter();
}
```

Constant time **->** Θ(1)

It just create and return it