GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 3 Report

Muhammed Sinan Pehlivanoğlu
1901042664

V0 TIME COMPLEXITY (USING ARRAY)

- CHECKING THE POSTION OF THE BUILDING TO BE LOCATED

```
protected boolean checkPosition(Building building){

    if((_rightSideNum == 0 ||_lefSideNum ==0) && building.getPosition()+building.getLenght() > _lenght){

        return  false;
    }

    for(int i = 0 ; i < _buildingNumber; ++i){
        if(building.getSide().equals(buildings[i].getSide())) {
            if(((buildings[i].getPosition()+buildings[i].getLenght())> building.getPosition()
            && building.getPosition()>= buildings[i].getPosition()) || building.getPosition()+building.getLenght() > _lenght)
            {
                return  false;
            }
        }
    }
    return true;
}
```

Best case is Ɵ(1). If the first condition is false or second condtion is false while querying first check.

Worst case is O(n). iterate as many as the number of buildings

- ENLARGING THE BUILDING ARRAY

```
private void enlargeBuilding(){
    Building[] tempBuildings = new Building[_buildingCapacity*2];
    System.arraycopy(buildings, srcPos: 0, tempBuildings,
            destPos: 0, _buildingNumber);
    buildings = tempBuildings;
    _buildingCapacity *=2;
}
```

The time complexity of Arraycopy function is O(n). As many as the **length** of the array copied. So time complexity **O(n).**

- ADDING THE BUILDING

```
public void addBuilding(Building building) {
    if(checkPosition(building)){
        if(_buildingNumber >= _buildingCapacity){
            enlargeBuilding();
        }
        buildings[_buildingNumber++] = building;
        building.setId(Building._nextId);
        ++Building._nextId;
        System.out.println(building.getClass().getSimpleName()+ " is added");
        if(building.getSide() == "r"){ ++_rightSideNum;}
        else if(building.getSide() == "l"){++_lefSideNum;}
    }
    else System.out.println("There is no enough space to locate the " + building.getClass().getSimpleName());
}
```

Best case is Ɵ(1) .if the array is not full.

Worst case is O(n) due to enlargeBuilding function.

- FINDING THE BUILDING WITH ID

```
protected Building findBuilding(int Id){
    for (int i = 0; i< getBuildingNumber(); ++i){
        if(buildings[i].getId() == Id){
            return  buildings[i];
        }
    }
    throw new NullPointerException("Building is not found");
}
```

Best case is Ɵ(1).

Worst case is O(n) lineer time.

- SHIFTING THE BUILDING ARRAY TO THE LEFT

```
private void shiftBuildingsArray(int index){

    for(int i = index ; i< _buildingNumber ; ++i){
        buildings[i] = buildings[i+1];
    }
}
```

Time complexity is O(n) Lineer time.

- TIDYING UP THE ARRAY

```java
private void tidyUptheArray(){
    if(_buildingNumber <= _buildingCapacity/4){
        Building[] temp = new Building[_buildingCapacity/2];
        System.arraycopy(buildings,  srcPos: 0, temp,
                 destPos: 0, _buildingNumber);
        buildings = temp;
        _buildingCapacity/=2;
    }
}
```

Best case is ϴ(1) constant time. if the length of the array is not equalte quarter of its capacity.

Worse case is ϴ(n) Lineer time Due to Copying the array with number of building.

- DELETING THE BUILDING

```java
public void deleteBuilding(int Id) {
    Building building = findBuilding(Id);
    for(int i =0 ; i< _buildingNumber; ++i){
        if(buildings[i].equals(building)){
            shiftBuildingsArray(i);
            --_buildingNumber;
            tidyUptheArray();
            System.out.println("Building is deleted");
            break;
        }
    }
}
```

Complexity of the **findBuilding method** is **O(n).**

Shifting the array takes O(N). So total complexiyy is O(N)+ O(N) = O(N).

- DISPLAYING THE BUILDING

```java
public void displayBuildings() {
    for (int i = 0; i< _buildingNumber; ++i){
        System.out.println(buildings[i]);
    }
}
```

Time complexity is **ϴ(N) lineer time.**

- REMAINIG LENGTH OF THE STREET

```java
public int remainigLenght() {
    int remainLenght = _lenght*2;

    for (int i = 0 ; i<_buildingNumber; ++i){
        remainLenght -= buildings[i].getLenght();
    }

    return remainLenght;
}
```

The complexity is **Ɵ(n) lineer time.**

- NUMBER and RATIO OF THE PLAYGROUND

```java
public void numberAndRatioOfPlayGround() {

    int totalLenghtPlayGround =0;
    int totalNumber =0;
    for (int i = 0 ; i<_buildingNumber; ++i){
        if(buildings[i] instanceof  PlayGround){
            totalLenghtPlayGround += buildings[i].getLenght();
            ++totalNumber;
        }
    }
    System.out.printf("Total Number : %d -> Ratio Of Playground:  %f \n",totalNumber,((float)totalLenghtPlayGround / (float) (_lenght*2)));
}
```

Best case is **Ɵ(1) if the Building is not PlayGround.**

Worst case is Ɵ(n) lineer time.

- TOTAL LENGTH OCCUPIED BY BUILDINGS

```java
public int totalLenghtOccupaided() {
    int occupiedLenght =0;

    for (int i = 0 ; i<_buildingNumber; ++i) {
        if (!(buildings[i] instanceof PlayGround)) {
            occupiedLenght += buildings[i].getLenght();
        }
    }
    return occupiedLenght;
}
```

Best case is **Ɵ(1) if the Building is not PlayGround.**

Worst case is Ɵ(n) lineer time.

- SWAP TWO BUILDING

```
private void swap(int index1 , int index2){
    Building temp = buildings[index1];
    buildings[index1] = buildings[index2];
    buildings[index2] = temp;
}
```

Time complexity is **ϴ(1).**

- **SORTING THE BUILDING ACCORDING TO ITS POSITIONS**

```
protected void sortBuildingPosition(){
    int minIndex =0;
    for(int i = 0; i< _buildingNumber ; ++i){
        for(int j = i+1 ; j<_buildingNumber ; ++j) {
            if (buildings[minIndex].getPosition() >= buildings[i].getPosition()) {
                minIndex = i;
            }
        }
        swap(minIndex,i);
    }
}
```

Time complexity is O(N^2).

- FILLING THE SILHOUTTE

```
protected void fillSilhoutte(){
    int max = findMaxHeight();

    for(int i = 0 ; i<max; ++i){
        for(int j = 0 ; j<_lenght ; ++j ){
            silhouette[i][j] = '.';
        }
    }
   for(int i = 0 ; i <_buildingNumber; ++i){
        int startHeight = max - buildings[i].getHeight();
        int endHeight = max;
        int endPosition = buildings[i].getLenght()+ buildings[i].getPosition();
        int startPosition = buildings[i].getPosition();

      for(int j = startHeight ; j< endHeight ; ++j){
        for(int k = startPosition ; k< endPosition ;++k  ){
            silhouette[j][k] = '#';
        }
      }
    }
}
```

First nested loop's time Complexity is **ϴ(max*length);**

Second nested loop's time Complexity is **ϴ(n*M*K); M = endHeigth, K = endPosition**

So the Time complexity is **ϴ(n*M*K).**

- CREATING THE SILHOUETTE

```
protected void createSilhoutte(){
    int maxHeight = findMaxHeight();
    sortBuildingPosition();
    silhouette = new char[maxHeight][_lenght];
    fillSilhoutte();

}
```

The comlexity of the sortBuildingPosition is **ϴ(max*length).**

The comlexity of the fillSilhoutte is **ϴ(N*M*K).**

So the time complexity is **ϴ(N*M*K**). **It depends on number of building , heigth of bulding and position of building.**

V1 TIME COMPLEXITY (USING ARRAYLIST)

- ADDING

```
public void addBuilding(Building building) {
    if(checkPosition(building)){
        _buildings.add(building);
        building.setId(Building._nextId);
        ++Building._nextId;
        ++_buildingNumber;
        System.out.println(building.getClass().getSimpleName()+ " is added");
        if(building.getSide() == "r"){ ++_rightSideNum;}
        else if(building.getSide() == "l"){++_lefSideNum;}
    }
    else System.out.println("There is no enough space to locate the " + building.getClass().getSimpleName());
}
```

Adding the element to array list takes **ϴ** (1) time in arrayList. But checkPosition method's time complexity is O(N). So time complexity O(N).

- DELETING

```
@Override
public void deleteBuilding(int Id) {
    Building building =  findBuilding(Id);
    _buildings.remove(building);
    --_buildingNumber;
}
```

Finding the building takes O(N). To remove by index, ArrayList find that index using random access in O(1) complexity, but after removing the element, shifting the rest of the elements causes overall **O(N)** time complexity.

- DISPLAYING

```
public void displayBuildings() {
    for (int i = 0; i< _buildingNumber; ++i){
        System.out.println(_buildings.get(i));
    }
}
```

The time complexity is O(N) . N is Number of the building.

- NUMBER AND RATIO OF PLAYGROUND

```
public void numberAndRatioOfPlayGround() {

    int totalLenghtPlayGround =0;
    int totalNumber =0;
    for (int i = 0 ; i<_buildingNumber; ++i){
        if(_buildings.get(i) instanceof  PlayGround){
            totalLenghtPlayGround += _buildings.get(i).getLenght();
            ++totalNumber;
        }
    }
    System.out.printf("Total Number : %d -> Ratio Of Playground:  %f \n",totalNumber,((float)totalLenghtPlayGround / (float) (_lenght*2)));

}
```

Best case is **Θ(1) if the Building is not PlayGround.**

Worst case is Θ(n) lineer time.

- TOTAL LENGTH OCCUPAIDED BY BUILDINGS

```
public int totalLenghtOccupaided() {
    int occupiedLenght =0;

    for (int i = 0 ; i<_buildingNumber; ++i) {
        if (!(_buildings.get(i) instanceof PlayGround)) {
            occupiedLenght += _buildings.get(i).getLenght();
        }
    }
    return occupiedLenght;
}
```

Best case is **Θ(1) if the Building is not PlayGround.**

Worst case is Θ(n) lineer time.

- REMAINIG LENGTH OF THE STREET

```
public int remainigLenght() {
    int remainLenght = _lenght*2;

    for (int i = 0 ; i<_buildingNumber; ++i){
        remainLenght -= _buildings.get(i).getLength();
    }

    return remainLenght;
}
```

The complexity is **Θ(n) lineer time.**

- CREATING SILHOUETTE

```
protected void createSilhoutte(){
    int maxHeight = findMaxHeight();
    sortBuildingPosition();
    silhouette = new char[maxHeight][_lenght];
    fillSilhoutte();
}
```

The comlexity of the sortBuildingPosition is **O(n*logn) due to use Collection.Sort() method.**

The comlexity of the fillSilhoutte is **O(N*M*K). M = heigth of building, k = position of building.**

So the time complexity is **O(N*M*K**). **It depends on number of building , heigth of bulding and position of building.**

V2 TIME COMPLEXITY (USING LINKEDLIST)

- ADD BUILDING

```
public void addBuilding(Building building) {
    if(checkPosition(building)){
        _buildings.add(building);
        building.setId(Building._nextId);
        ++Building._nextId;
        ++_buildingNumber;
        System.out.println(building.getClass().getSimpleName()+ " is added");
        if(building.getSide() == "r"){ ++_rightSideNum;}
        else if(building.getSide() == "l"){++_lefSideNum;}
    }
    else System.out.println("There is no enough space to locate the " + building.getClass().getSimpleName());
}
```

Adding the element to the last of Linkedlist takes ϴ (n) time. checkPosition method's time complexity is alse O(N). So time complexity O(N).

- SORTING

```
protected void sortBuildingPosition() { Collections.sort(_buildings); }
```

First the LinkedList is converted to array. It takes **O(N)** time. Then the Collection.sort algorithm takes **O(NlogN)** time due to Merge sort. So total complexity is **O(Nlogn).**

- DELETE

```
@Override
public void deleteBuilding(int Id) {
    Building building =  findBuilding(Id);
    _buildings.remove(building);
    --_buildingNumber;
}
```

Remove method of Linked list takes **O(N)** time. FindBuilding is also **O(N)** time. So the complexity is **O(N).**

- DISPLAYING

```java
public void displayBuildings() {
    for (int i = 0; i< _buildingNumber; ++i){
        System.out.println(_buildings.get(i));
    }
}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is $O(N^2)$.

- NUMBER AND RATIO OF PLAYGROUND

```java
public void numberAndRatioOfPlayGround() {

    int totalLenghtPlayGround =0;
    int totalNumber =0;
    for (int i = 0 ; i<_buildingNumber; ++i){
        if(_buildings.get(i) instanceof  PlayGround){
            totalLenghtPlayGround += _buildings.get(i).getLenght();
            ++totalNumber;
        }
    }
    System.out.printf("Total Number : %d -> Ratio Of Playground:  %f \n",totalNumber,((float)totalLenghtPlayGround / (float) (_lenght*2)));

}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is $O(N^2)$.

- TOTAL LENGTH OCCUPAIDED BY BUILDINGS

```java
public int totalLenghtOccupaided() {
    int occupiedLenght =0;

    for (int i = 0 ; i<_buildingNumber; ++i) {
        if (!(_buildings.get(i) instanceof PlayGround)) {
            occupiedLenght += _buildings.get(i).getLenght();
        }
    }
    return occupiedLenght;
}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is $O(N^2)$.

.

- REMAINIG LENGTH OF THE STREET

```
public int remainigLenght() {
    int remainLenght = _lenght*2;

    for (int i = 0 ; i<_buildingNumber; ++i){
        remainLenght -= _buildings.get(i).getLenght();
    }

    return remainLenght;
}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is $O(N^2)$.

- CREATING SILHOUETTE

```
protected void createSilhoutte(){
    int maxHeight = findMaxHeight();
    sortBuildingPosition();
    silhouette = new char[maxHeight][_lenght];
    fillSilhoutte();
}
```

The comlexity of the sortBuildingPosition is **O(n*logn) due to use Collection.Sort() method.**

The comlexity of the fillSilhoutte is **O(N*M*K). M = heigth of building, k = position of building.**

So the time complexity is **O(N*M*K**). **It depends on number of building , heigth of bulding and position of building.**

V3 TIME COMPLEXITY (USING LDLINKEDLIST)

- ## ADDING THE HEAD OF LDLINKEDLIST

```
private void addFirst(E element){
    head =  new Node<E>(element,head);
    ++size;
}
```

It takes ϴ(1) time.

- ## ADDING THE NODE AFTER THE REFERENCED NODE

```
private void addAfter(E element, Node<E> node){
    node.next = new Node<E>(element,node.next);
    ++size;
}
```

It takes **ϴ(1) time.**

- ## LINKING NODE FROM LIST STORE REMOVED NODES TO ORIGINAL LIST

```
private boolean link(E element){
    Node<E> temp;
    int index = removed.indexOf(element);


    if(index == -1) return  false;

    else if(index == 0){
        temp = removed.head;
        removed.head = removed.head.next;
        getNode( index: size-1).next = temp;
        temp.next = null;
        --removed.size;
        ++size;
        return true;
    }
    else{
        temp = removed.getNode( index: index-1);
        Node<E> finded = temp.next;
        temp.next = temp.next.next;
        --removed.size;
        getNode( index: size-1).next= finded;
        finded.next =null;
        ++size;
        return true;
    }
}
```

Best case is **ϴ(1) time. If the element is not found at LDLinkedList.**

Worse case is O(N). If the element is at the end of the List. So GetNode method takes O(N) time.

- # ADD ELEMENT TO LIST

```
@Override
public  boolean add(E element){

    if(head == null){
        if(!link(element)){
            addFirst(element);
        }
    }
    else {
         if(!link(element)) {
            Node<E> last = getNode( index: size - 1);
            addAfter(element, last);
        }
    }
    return  true;
}
```

Link method takes O(N).AddFirst and Addlast takes **Ɵ(1) time. So total complexity is O(N) lineer time.**

- # GET NODE

```
private Node<E> getNode(int index){
    Node<E> iter = head;
    for(int i = 0;  i< index && iter != null ; ++i){
        iter = iter.next;
    }

    return  iter;
}
```

It takes **O(N) time to find desired Node.**

- # ADDING NODE TO LIST STORE REMOVED NODE

```
private void addNodetoRemoved(Node<E> node){
    if(removed.head == null){
        removed.head = node;
    }
    else{
        removed.getNode( index: removed.size-1).next = node;
    }
    ++removed.size;
}
```

Best case : **Ɵ(1) if the list of removed element's head is null.**

Worst case : O(N) due to getNode method.

- ## REMOVING FIRST NODE FROM LIST

```
private E removeFirst(){
    Node<E> temp = head;
    if (head != null){
        head = head.next;
    }
    if (temp != null) {
        --size;
        addNodetoRemoved(temp);
        return temp.data;
    }
    throw new NoSuchElementException("The List is Empty");
}
```

Best case : Θ(1) if the head is null.

Worst case : O(N). Due to addNodeToremoved takes O(N). N is node number of removed List.

- ## REMOVING THE NODE AFTER THE REFERENCED NODE

```
private E removeAfter(Node<E> node) {
    Node<E> temp = node.next;
    if (temp != null) {
        node.next = temp.next;
        size--;
        addNodetoRemoved(temp);
        return temp.data;
    }
    throw new NoSuchElementException("The List is Empty");
}
```

Best case : Θ(1) if the temp node is null.

Worst case : O(N). Due to addNodeToremoved takes O(N). N is node number.

- # REMOVING ELEMENT FROM LIST

```
*/
public E removee(E element){

    if(head.data.equals(element)){

        return  removeFirst();
    }
    else{
        return removeAfter(getNode( index: indexOf(element) - 1));
    }
}
```

Remove first and remove after methods takes O(N) time. GetNode also takes O(N) time .

So Total complexity is O(N) lineer time.

- GET DATA

```
*/
@Override
public E get(int index) {
    if(index >= size){

        throw new IndexOutOfBoundsException("Invalid Index");
    }
    Node<E> get = getNode(index);

    return  get.data;
}
```

Best case : **Ɵ(1)**

Worst Case : O(N) due to getNode method .

- DISPLAYING

```java
public void displayBuildings() {
    for (int i = 0; i< _buildingNumber; ++i){
        System.out.println(_buildings.get(i));
    }
}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is O(N²).

- NUMBER AND RATIO OF PLAYGROUND

```java
public void numberAndRatioOfPlayGround() {

    int totalLenghtPlayGround =0;
    int totalNumber =0;
    for (int i = 0 ; i<_buildingNumber; ++i){
        if(_buildings.get(i) instanceof  PlayGround){
            totalLenghtPlayGround += _buildings.get(i).getLenght();
            ++totalNumber;
        }
    }
    System.out.printf("Total Number : %d -> Ratio Of Playground:  %f \n",totalNumber,((float)totalLenghtPlayGround / (float) (_lenght*2)));

}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is O(N²).

- TOTAL LENGTH OCCUPAIDED BY BUILDINGS

```
public int totalLenghtOccupaided() {
    int occupiedLenght =0;

    for (int i = 0 ; i<_buildingNumber; ++i) {
        if (!(_buildings.get(i) instanceof PlayGround)) {
            occupiedLenght += _buildings.get(i).getLenght();
        }
    }
    return occupiedLenght;
}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is $O(N^2)$.

.

- REMAINIG LENGTH OF THE STREET

```
public int remainigLenght() {
    int remainLenght = _lenght*2;

    for (int i = 0 ; i<_buildingNumber; ++i){
        remainLenght -= _buildings.get(i).getLenght();
    }

    return remainLenght;
}
```

The time complexity of get Method is O(N). Get method Works N = buildingNumber.

So total Complexity is $O(N^2)$.

- CREATING SILHOUETTE

```
protected void createSilhoutte(){
    int maxHeight = findMaxHeight();
    sortBuildingPosition();
    silhouette = new char[maxHeight][_lenght];
    fillSilhoutte();
}
```

The comlexity of the sortBuildingPosition is **O(n*logn) due to use Collection.Sort() method.**

The comlexity of the fillSilhoutte is **O(N*M*K). M = heigth of building, k = position of building.**

So the time complexity is **O(N*M*K**). **It depends on number of building , heigth of bulding and position of building.**

NODE CLASS

```java
 */
static class Node<E>{
    Node next;
    E data;
    Node(E element){
        data =element;
        next =null;
    }
    Node(E element, Node inserted){
        data = element;
        next = inserted;
    }

    private E get(){
        return data;
    }

}
```

PRIVATE ITERATOR CLASS METHODS

- CHECKING IF THE NEXT NODE OF THE LIST IS NULL OR NOT.

```java
@Override
public boolean hasNext() {

    if(iter.next == null){

        return false;
    }
    else{

         return  true;
    }

}
```

IT TAKES **ϴ(1).**

- RETURN NEXT ELEMENT OF THE LIST

```
@Override
public E next() {
    if(!hasNext()){
        throw new  NullPointerException("List is empty");
    }
    else{
        iter = iter.next;
        return  iter.data;
    }
}
```

IT TAKES **ϴ(1).**

- CONSTRUCTOR OF LDITERATOR. IT IS PRIVATE.

```
// Creating a new class that implements the interface Iterator.
private class LdIterator<E> implements Iterator<E>{
    private Node<E> iter;
    public LdIterator(){
        iter = (Node<E>) head;
    }
```

IT TAKES **ϴ(1).**

- REMOVE METHOD OF ITERATOR

```
@Override
public void remove(){
    int index = indexOf(iter);
    LDLinkedList.Node<E> temp = (LDLinkedList.Node<E>) getNode( index: index-1);
    temp.next = iter.next;
    iter.next = null;
}
```

IT TAKES **O(N). DUE TO GETNODE TAKES O(N).**

- TO STRING OVERRIDEN METHOD

```
public String toString() {
    return iter.get().toString();
}
```

IT TAKES **ϴ(1).**

OVERRIDEN ITERATOR METHOD

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

IT TAKES **Θ(1). RETURN ITERATOR.**