**SINGLE-LINKED LISTS**

ArrayList is limited bc add and remove methods operate in linear (O(n)) time, requiring a loop to shift elements.

A linked list can add and remove elements at a known location in O(1) time.

How you add to position n/2 (n: size of the linked list)?

You have to start from the beginning and traverse until (n/2)th node. This will take linear time but we claimed that it happens in constant time. It is only possible if you know the location, not the index.

How we know the location?

We have a concept called iterator. We are gonna use the add and remove operations of an iterator. Since iterator points to a node, we can perform these 2 operations in constant time.

🡪 Iterators point somewhere in between the elements.

A List Node

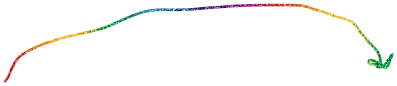
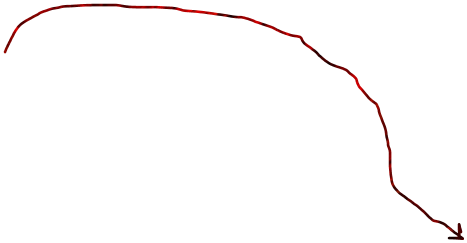
Diagram

Description automatically generatedA node can contain:

* a data item
* one or more links
  + link is reference to a list node

In our structure, the node contains a data field named data of type E and a reference to the next node, named next

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generatedText, letter

Description automatically generated

Node will be the inner class of LinkedList class. It is not gonna access the members of linked list since it is static.

Node class is private so it cannot be used by anything outside the linked list class.

We don’t want anybody to access our Node structure and modify our LinkedList structure.

Diagram

Description automatically generated

Node<String> tom = new Node<String>(“Tom”);

Node<String> dick = new Node<String>(“Dick”);

Node<String> harry = new Node<String>(“Harry”);

Node<String> sam = new Node<String>(“Sam”);

tom.next = dick;

dick.next = harry;

harry.next = sam;

Generally, we don’t have individual references to each node.

A SingleLinkedList object has a data field head, *the list head*, which references the first list node.

Text

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

SLList.addFirst(E item)

Diagram

Description automatically generated

We created 1 node and performed 2 link updates. That requires constant time.

private void addFirst( E item ){

Node<E> temp = new Node<E>(item, head);

head = temp;

size++;

}

OR

private void addFirst( E item ){

head = new Node<E>(item, head);

size++;

}

Actually, you can make this method public and present it as additional method to SLList or it can be method in the List interface and you can consider as implementation of that method in List interface.

addAfter(Node <E> node, E item)

Graphical user interface, diagram, application

Description automatically generated

This method cannot be public since this node structure is not known outside SLList class, node is private. Any method that using node structure as a parameter cannot be public.

This method also requires constant time.

private void addAfter( Node<E> node, E item ){

Node<E> temp = new Node<E>(item, node.next);

node.next = temp;

size++;

}

OR

private void addAfter( Node<E> node, E item ){

node.next = new Node<E>(item, node.next);

size++;

}

removeAfter(Node<E> node)

Graphical user interface, diagram

Description automatically generated

Graphical user interface, diagram, application

Description automatically generated

If you want to return the removed data, you can keep that node as temp.

Running time is constant.

private E removeAfter(Node<E> node) {

Node<E> temp = node.next;

if (temp != null) {

node.next = temp.next;

size--;

return temp.data;

}

else{

return null;

}

}

removeFirst()

Graphical user interface, diagram

Description automatically generated

Graphical user interface, diagram

Description automatically generated

Running time is constant.

private E removeFirst() {

Node<E> temp = head;

if (head != null) {

head = head.next;

}

if (temp != null) {

size--;

return temp.data;

}

else {

return null;

}

}

TRAVERSING A SINGLE-LINKED LIST

Graphical user interface, diagram

Description automatically generated

toString() can be implemented with a traversal:

public String toString(){

Node<String> nodeRef = head;

StringBuilder result = new StringBuilder();

while (nodeRef != null) {

result.append(nodeRef.data);

if (nodeRef.next != null) {

result.append(“ ==> “);

}

nodeRef = nodeRef.next;

}

return result.toString();

}

Running time is linear time **(θ(n)).**

We use StringBuilder not String because String is inmutable. Only thing you can do is copy 2 strings by concatenation to form another string.

Each time you append new string at the end of another string, you have to copy n times. Running time is a lot slower.

Each data field has constant length (3 or 4, you can assume at most 10). StringBuilder keeps references of all the strings that you put and at the end, convert to a string. What it does is to reserve a memory space that is enough for the string and put : “Tom ==> Dick ==> Ann”, to that reserved space.

What if you use String?

First “Tom” will be copied.

Tom

Tom ==>

Tom ==> Dick

Tom ==> Dick ==>

Tom ==> Dick ==> Ann

Then at the end of Tom, “ ==> ” will be copied.

Then “Dick” will be copied.

Then “ ==> “ will be copied.

Then “Ann” will be copied.

All these strings are generated. That requires quadratic time **(θ(n^2)).**

**VERY IMPORTANT !!!!!!**

getNode(int)

In order to implement methods required by the List interface, we need an additional helper method:

private Node<E> getNode(int index){

Node<E> node = head;

for (int i = 0; i < index && node != null; i++) {

node = node.next;

}

return node;

}

Tb(n) = θ(1)

Tw(n) = θ(n)

T(n) = O(n)

Methods that should be implemented in SingleLinkedList class:

Table

Description automatically generated

public E get (int index){

if (index < 0 || index >= size) {

throw new

IndexOutOfBoundsException(Integer.toString(index));

}

Node<E> node = getNode(index);

return node.data;

}

T(n) = O(n)

public E set (int index, E anEntry) {

if (index < 0 || index >= size) {

throw new

IndexOutOfBoundsException(Integer.toString(index));

}

Node<E> node = getNode(index);

E result = node.data;

node.data = anEntry;

return result;

}

T(n) = O(n)

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);

}

}

Tb(n) = θ(1)

Tw(n) = θ(n)

T(n) = O(n)

We can simply write add. We have to get to the last element. It is worst case of getNode method so it is θ(n) not O(n). We use getNode method to get reference of last element and perform addAfter.

getNode takes θ(n) time, addAfter takes constant time, total is θ(n).

To add an item to the end of the list:

public boolean add (E item) {

add(size, item);

return true;

}

If we keep the tail as well, we can make adding to end constant time.