Linked List

Introduction

> **Data Structure$**: Variables that store information used by algorithms

- Ideal data structure:
  - Fast insertion (when adding a new item to data structure)
  - Fast deletion (when removing an item from the data structure)
  - Fast lookup
  - Minimum memory usage
- Unfortunately: there is no ideal data structure.
- Two basic data structures:
  - Array
  - Linked List
- Array:
  - Array is a series of variable that
    - are of the same data type and
    - are stored in consecutive memory locations
  - The memory used to store an array must be allocated up front
  - Strength: Fast access using an array index (because the memory address of `x[i]` can be computed easily).
  - Weakness of an array: cannot increase the array size
    - To increase the array, we must use memory cells that follows the last element of the array.
    - However, these memory cells may not be available (since they are used by another variable).
  - Weakness of an array: it takes a long time to insert a value in the middle of an array.
    - We have to shift many elements over.
- The linked list data structure:
  - The linked list and array are complementary to each other.
  - Characteristics of a linked list:
    - Each list element is allocated individually (and then linked into the list)
      - Memory for all array elements are allocated up front
    - It's easy to insert/delete elements from a linked list

        - It's hard to insert/delete elements from an array
    - It is slow to look up elements in a linked list by its index
        - It is fast to look up elements in an array by its index
- A linked list consists of a chain of list objects. A list object is often called a **node**.
- Every node consists of two parts:
    - One of more data fields (contain the information stored in the linked list)
    - A link (reference variable) (contains the reference (=address) of the next node/list element).
        - The link in the last node is `null` (=end of the list).
- A Java program will have a reference variable (commonly named as `head` or `first`) that contains the reference to the first node (=list element).
    - Consequently, only the data stored in the first node is immediately accessible.
    - Accessing the data stored in the other nodes will reply on the reference stored in the first node.

```
public class Node{
    int item; // int data stored in the Node
    Node next; // Link that reference to the next node
}
```

- Define a `Node` class for a linked list:
    - The class `Node` contains a reference variable `next` that references to a `Node` (same class) object.
    - The `next` variable is used to create a chain of Nodes
    - We can define other data field depending on what we want to store in a Node.

Array	Linked List
Array elements are stored contiguously in memory	List elements (=nodes) do not need to be stored contiguously in memory
All array elements are allocated at once	Nodes can be allocated piece meal when needed
Once allocated, the number of elements in the array is fixed	We can increase the number of elements in a list easily by increasing the length of the chain
Only store data fields, do not need to store non-data fields	Requires the use of a linking field (`next`) to create a chain
Accessing the k-th element in an array is	Need to traverse the chain to reach the k-th

Array	Linked List
fast	element - slow
Inserting a value in the middle of an array is difficult (need to shift elements over)	Inserting a node in the middle of a linked list is easy

Implementing a Simple Linked List

- Operations on a Linked List
  - The linked list is a data structure used to store information.
  - To be useful as a data structure, the linked list must support the following operaqtions:
    - Create an empty linked list
    - Insert a data item into the linked list
      - Insert it at the beginning of the linked list
      - Insert it at the end of the linked list
    - Delete a data item from the linked list
      - Insert the element at the beginning of the linked list
      - Insert the element at the end of the linked list
    - Searching for some data item in the linked list
      - Search by its index
      - Search by its key value
- Constructing a linked list using explicit helper reference variables

```
public class Demo {
    public static Node first; // Define the first variable

    public static void main(String[] args) {
        Node help1 = new Node();
        Node help2 = new Node();
        Node help3 = new Node();

        help1.item = "to";
        help2.item = "be";
        help3.item = "or";

        // Create the chain
        help1.next = help2;
        help2.next = help3;
        help3.next = null;

        first = help1;
    }
}
```

Standard List Traversal Algorithm

```
// The standard list raversal algorithm
Node p;
p = first; // p now points to the first node

while (p != null) {
    // process data p.item in node p
    p = p.next; // advances p to next node
}
```

- Example: Use the list traversal algorithm to print out all items in a list:

```
Node current;
current = first; // initialize current to the first node
while (p != null) {
    System.out.println(current.item);
    current = current.next;
}
```

- Enhancements of the list traversal algorithm: find a node that contains `X`.

```
Node current = first;
while (current != null && !current.item.equals("X")) {
    current = current.next;
}
```

- Use the list traversal algorithm to find a list element that contains a certain key

```
/**
 * This method returns a list element that contains a certain value
 * @param f the linked list to be searched
 * @param s the value of searching
 * @return the node containing s
 */
public static Node findNode(Node f, String s) {
    Node current = f;
    while (current != null) {
        if (current.item.equals(s)) { // found!
            return current;
        }
        current  = current.next;
    }
    return null; // not found
}
```

- List traversal using a for-loop: print out item in the linked list using a for-loop:

```
for (Node p = first; p != null; p = p.next) {
    System.out.println(p.item);
}
```

Implementing the Standard Operations on a Simple Linked List

- Operations on a linked list:
  - Test if a list is empty
  - Get the item at the start
  - Get the item at the end
  - Insert an item at the start
  - Insert an item at the end
  - Delete the item at the start
  - Delete item at the end
  - Get the item at position `k` in the linked list
  - Remove the first item that contains the value key from the linked list

```
public interface SimpleList<T> { // A list that store objects of type T
    public boolean isEmpty(); // check if the list is empty

    public T getFirst(); // get the first item in the list
    public T getLast(); // get the last item in the list

    public void addFirst(T item); // add an item to the beginning of the list
    public T removeFirst(); // remove the first item from the list

    public voild addLast(T item); // add an item to the end of the list
    public T removeLast(); // remove the last item from the list

    public T get(int pos); // get the item at position pos in the list
    public void remove(T key); // remove the first item that contains the value key fro
}
```

```java
import java.util.NoSuchElementException;

public class GenericLinkedList<T> implements SimpleList<T> {
    // Start of the inner class Node
    public class Node<T> {
        // Node instance variables
        private T item; // the data stored in the node
        private Node<T> next; // the link to the next node

        // Constructor for Node
        public Node(T item, Node<T> next) {
            this.item = item;
            this.next = next;
        }

        public String toString() {
            return "" + this.item;
        }
    } // End of the inner class Node

    // The list starts at first
    private Node<T> first;

    // Contructs an empty list
    public GenericLinkedList() {
        first = null;
    }

    // check if the list is empty
    public boolean isEmpty(){
        return first == null;
    }

    // get the first item in the list
    public T getFirst() {
        // Edge case: list is empty
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        return first.item; // retrieves item in first node
    }
    // get the last item in the list
    public T getLast() {
```

```
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    // (1) Find the last node
    Node<T> current = first;
    while (current.next != null) {
        current = current.next;
    }
    // (2) Return item stored in this node
    return current.item;
}

// add an item to the beginning of the list
public void addFirst(T item) {
    Node<T> = newNode = new Node<>(item, first);
    first = newNode;
}
// remove the first item from the list
public T removeFirst() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    T toReturn = first.tiem // first = null -> crash
    first = first.next;
    return toReturn;
}

// add an item to the end of the list
public voild addLast(T item) {
    if (isEmpty()) { // edge case
        first = new Node<>(item, null);
        // equivalent to: addFirst(item)
    } else {
        // (1) Find the last element in the linked list
        Node<T> current = first; // if first = null -> crash
        while (current.next != null) { // Find the last element -> list traversal
            current = current.next;
    }
    // (2) Mkae a new Node with data to linke to the last element
    current.next = new Node<>(item, null);
    }
}
// remove the last item from the list
```

```
public T removeLast() {
    // Edge case 1: list is empty
    if (isEmpty()) { // empty list cannot remove anything
        throw new NoSuchElementException();
    }
    // Edge case 2: list only contain one element
    if (first.next == null) {
        Node<T> ret = firtst; // save for return
        first = null; // unlink by updating first
        return ret.item;
    }
    //General Case: list not empty and have at least 2 elements
    // to remove last, we need information of the second last element
    Node<T> previous = first;
    Node<T> current = first;

    while (current.next != null) { // find the last list element
        previous = current; // keep track of the previous node
        current = current.next;
    }
    // previous points to the predecessor node of the last node
    // current points to the last node
    // Unlink the last node
    previous.next = null;
    // return item in the last node
    return current.item;
}

// get the item at position pos in the list
public T get(int pos) {
    // Edge case
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    // General case
    int i = 0;
    Node<T> current = first;
    while (current != null) {
        if (i == pos) {
            break;
        }
        i++;
        current = current.next;
```

```
        }
        if (current == null) {
            throw new IndexOutOfBoundsException();

        }
        return current.item;
    }
    // remove the first item that contains the value key from the list
    public void remove(T key) {
       // Edge case 1: list is empty
       if (first == null) {
           throw new NoSuchElementException();
       } else if (first.item.equals(key)) {
           // Edge case 2: the first item is the key
           first = first.next
       } else {
           // General case
           // (1) Find the predecessor node of the node containing item == key
           Node<T> = current = first; // Initialize
           Node<T> = previous = first;
           while (current != null && !current.item.equals(key)) {
               previous = current;
               current = current.next;
           }
           if (current == null) {
               // key not found
               throw new NoSuchElementException ();
           }
           // (2) Unlink the targeted node from its predecessor node
           previous. next = current.next;
       }
    }
}
```

- It is important to consider edge cases when defining methods.
  - Often, the edge cases are:
    - The empty list: `first == null`
    - A list contains onlly 1 element: `first == null`
- Garbage: Note that we origianl first node is not referenced to by any permanent variables after the operation `removeFirst()`
  - Such objects are known as garbage
  - Objects that become garbage are inaccessible and unusable in the program

Other types of Simple Linked List

- Simple Linked List
  - The simple (or singly-cahined) linked list is chained linearly and without a loop
  - A variant of the simple linked list is double-ended list. It uses a `last` reference to help find the last element quickly.
  - The simple circular linked list is chained linearly and contains a loop.
- Doubly Linked List
  - The doubly linked list is a chained linearly using forward and backward links without a loop.
  - The doubly linked circular list is chained using forward and backward links and contains two loops

Using a Linked List in a `for-each` Loop - Implementing the `Iterable` Interface and the `Iterator` Interface

- Java has a `for-each` loop that iterates over collection objects:

```
for (dataType x : Collection) {
    // operations on x
}
```

- For example, an `ArrayList` is an iterable collection

```
ArrayList<Integer> a = new ArrayList<>();
a.add(1); a.add(2); a.add(3);

for (Integer i : a) {
    System.out.println(i);
}
```

- We can use an `ArrayList` in the `for-each` loop because the `ArrayList` is `Iterable`. Our previous implementation of the `GenericLinkedList` class is not iterable, so we cannot use it in the `for-each` loop.
- Now, we make some changes to the `GenericLinkedList` class so that it is iterable:

```
public class GenericLinkedList<T> implements SimpleList<T>, Iterable<T> {
    // Private inner class Node
    private class Node<T> {
        private T item;
        private Node<T> next;

        // ...
    }

    // Private inner class MyLinkedListIterator
    private class MyLinkedListIterator<T> implements Interator<T> {
        private Node<T> current; // current position in the iteration

        public MyLinkedListInterator(Node<T> f) { // contructor
            current = f; // initialize
        }

        public boolean hasNext() {
            // returns true if there are more nodes
            return current != null;
        }

        public T next() {
            // return the next element in the iteration
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            T res = current.item; // get the current item
            current = current.next; // advance to the next node
            return res;
        }
    }
    private Node<T> first;
    public GenericLinkedList() {
        first = null;
    }
    // ... Other methods
    public Iterator<T> iterator() { // implements the Iterable interface
        return new LinkedListIterator<T>(first);
    }
}
```

Use Linked List to Implement a Stack

```
public class ListStack<T> implements MyStack<T> {
    // private inner class Node
    prviate class Node<T> {
        private T item;
        private Node<T> next;

        public Node(T item, Node<T> next) {
            this.item = item;
            this.next = next;
        }

        public String toString() {
            return "" + this.item;
        }
    } // End of the private inner class Node

    private Node<T> frist;

    public ListStack() { // constructor
        first = null; //empty list = empty stack
    }

    // returns true is stack is empty
    public boolean isEmpty() {
        return first == null;
    }

    // returns true if stack is full
    public boolean isFull() {
        return false; // linked list is not fixed size
    }

    // push(item) -> insert at the front of the list
    public void push(T item) {
        Node<T> newNode = new Node<T>(item, first);
        first = newNode;
    }

    // pop() -> remove the first item and return it
    public T pop() {
```

```
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        T toReturn = first.item;
        first = first.next;
        return toReturn;
    }

    // peek() -> return the first item
    public T peek() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        return first.item;
    }
}
```

- Properties of the stack implementation using a linked list
  - The `push()` and `pop()` does not have any loops!
    - We say that the `push()` and `pop()` runs in constant time
  - We can also implement a stack using `push(item)=addLast(item)` and `pop()=removeLast()`.
    - However, this is not preferred because `addLast()` and `removeLast()` requires the use of the list traversal algorithm and will run slower.

Implementing Queue Using a Linked List

```
public class ListQueue<T> implements MyQueue<T> {
    // private inner class Node
    private class Node<T> {
        private T item;
        private Node<T> next;

        public Node(T item, Node<T> next) {
            this.item = item;
            this.next = next;
        }

        public String toString() {
            return "" + this.item;
        }
    } // End of private inner class

    private Node<T> first;

    public ListQueue() { // constructor
        first = null; // empty list = empty queue
    }

    // returns true if queue is empty
    public boolean isEmpty() {
        return first == null;
    }

    // returns true if queue is full
    public boolean isFull() {
        return false; // linked list is not fixed size
    }

    // enqueue(item) -> insert at the end of the list
    public void enqueue(T item) {
        if (isEmpty()) {
            first = new Node<T>(item, null);
        } else {
            Node<T> current = first;
            while (current.next != null) {
                current = current.next;
```

```
            }
            current.next = new Node<T>(item, null);
        }
    }

    // dequeue() -> remove the first item and return it
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        T toReturn = first.item;
        first = first.next;
        return toReturn;
    }

    // peek() -> return the first item
    public T peek() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        return first.item;
    }
}
```

- Postscript:
  - We can also implement a queue using:
    - `enqueue(item) = addFirst(item)`
    - `dequeue() = removeLast()`

Java's `LinkedList<E>` Class

- Java's `LinkedList<E>` class implemented as a doubly-linked list
- Sample methods:

```
size() // returns the number of elements in this list

addFirst(E e);
addLast(E e);
add(int index, E element)

removeFirst();
removeLasT();
remove(int index)

getFirst();
getLast();
get(int index)
```