CSCI 2110 Data Structures and Algorithms

Module 3: Unordered Lists

What we will discuss in this module

- Data Structures: The Big Picture
- UNORDERED LISTS – our first data structure
  - Definition and Examples
  - Choices for Implementation
  - Implementation
  - Application
  - Analysis

What is a data structure?

A data structure is a *collection* of data items, that, together, constitute a meaningful whole.

A data structure has a *state* (what it contains) and *behaviour* (what operations can be performed).

Thus, in object-oriented programming, a data structure can be implemented as a *class file*.

A data structure is composed of (can be built from) other smaller data structures

Example

- Book - Collection of pages
- Page - Collection of lines
- Line - Collection of characters
- Character - Collection of pixels
- Pixel - Collection of lithographic units for printing

A abstract data type (ADT) defines a data structure without referring to its implementation details

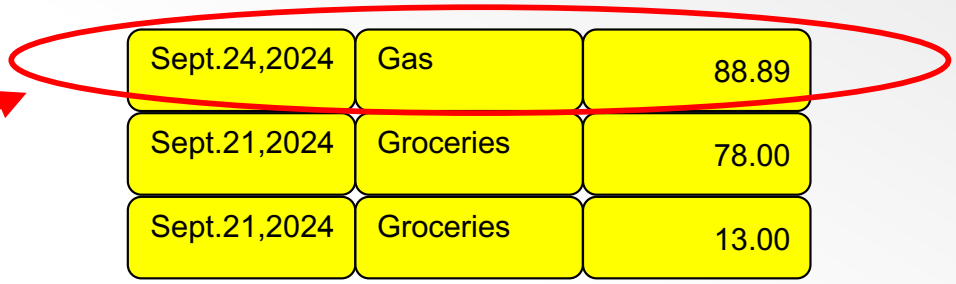
Example:
A Stack ADT

- constructor – Create an empty stack
- push – Add an item to the top of the stack (if the stack is not full)
- pop – Remove an item from the top of the stack (if the stack is not empty)
- peek – Retrieve the item at the top of the stack without removing it (if the stack is not empty)
- isEmpty – Return true if the stack is empty
- isFull – Return true if the stack is full

Unordered Lists – our first data structure

An unordered list is a simple linear collection of data items.
Relative positions of the items is irrelevant.
Items can also be repeated.

Example:
List of expense items



Each item is an **Expense object**,
defined by String date, String description, and double amount

Another example: List of inter-city distances (From, To, Distance)

Let's call our Unordered List as the List class. What are the operations in this class?

Operations that we want in our List class

- *Add an item to the list.*
- *Remove a specific item from the list.*
- *Step through all the items.*
- ***<*IMPORTANT*> Search the list to determine whether it contains a specific item.***

Other useful operations

- *Get the size of the list.*
- *Check if the list is empty.*
- *Clear the list.*
- *Remove all occurrences of a specific item from the list.*
- *Get the first item from the list.*
- *Get the next item from the list.*

What is the complexity of the most important operation (search)?

It would be O(n) worst case complexity, where n is the number of items.

Reason:

- *Since the list is unordered, the relative positions of items are irrelevant.*
- *Hence each search requires a scan from the first item until the item is found or until the end of the list is reached.*
- *If n is the number of items, such a search would require O(n) time in the worst case.*

Such a List class can then be used by different applications, such as an Expense List class.

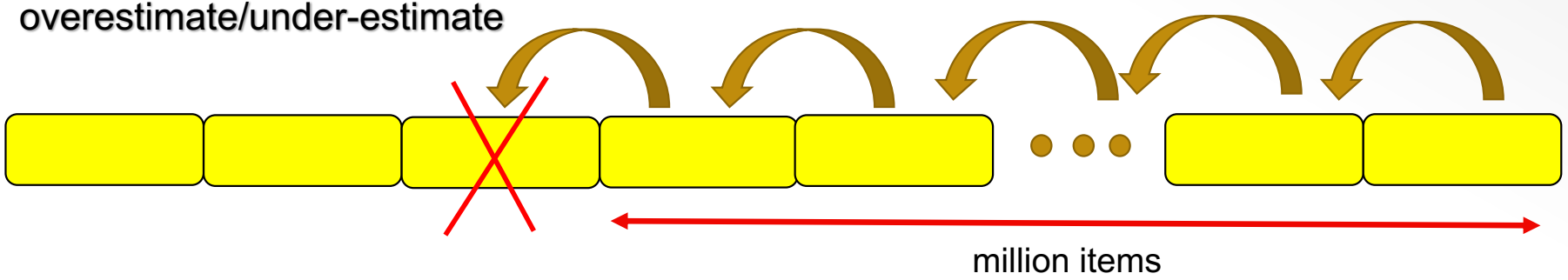
The Expense List class would have appropriate operations to answer the following types of questions:

- *What is the maximum (or minimum) expense, and on what item?*
- *What is the average expense?*
- *What is the total amount spent on a given item?*
- *What is the item-wise breakup of expenditures?*
- *What is the total amount spent in a given time period?*
- *etc.*

How do we build the List class? Let's look at our choices for implementation

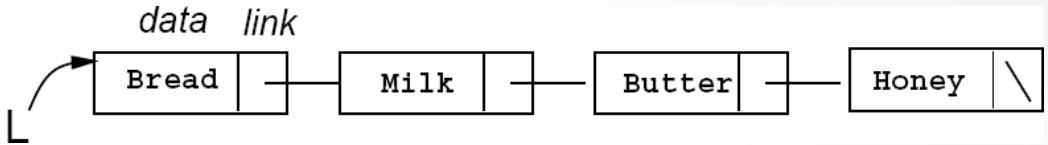
Arrays
ArrayLists
Linked Lists

Problem with Arrays and ArrayLists:
They are position-based data structures – successive items are stored in consecutive memory locations.
A random deletion or insertion of an item can cause large numbers of items to be moved.
Another drawback in an array is that storage is allocated in one shot, so we may overestimate/under-estimate



What about a Linked List?

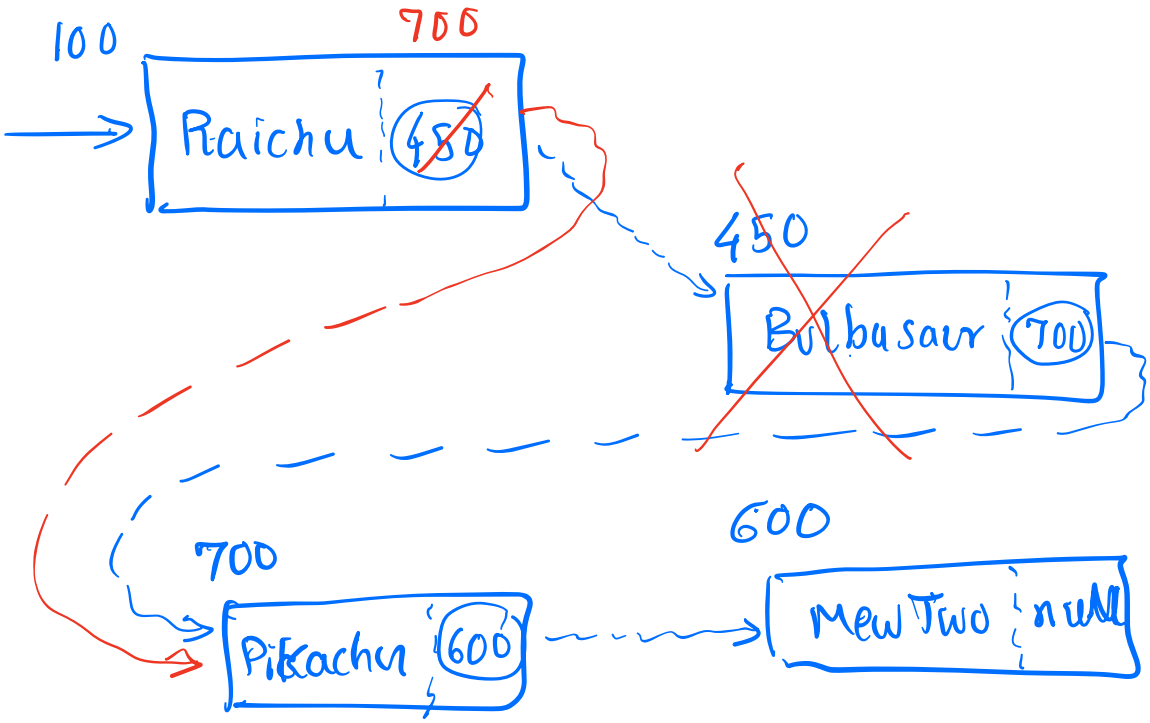
A linked list is a linear structure consisting of ***nodes***.
Each node has a ***data part*** that holds the client-supplied information,
and a ***link*** or a reference to the next node.



To access the entries of the linked list, a reference to its first entry is all we need.

One can access any entry by simply following the chain of links.
When an entry is removed from some place in a linked list,
all that needs to be done is to have its predecessor's link refer to its successor.

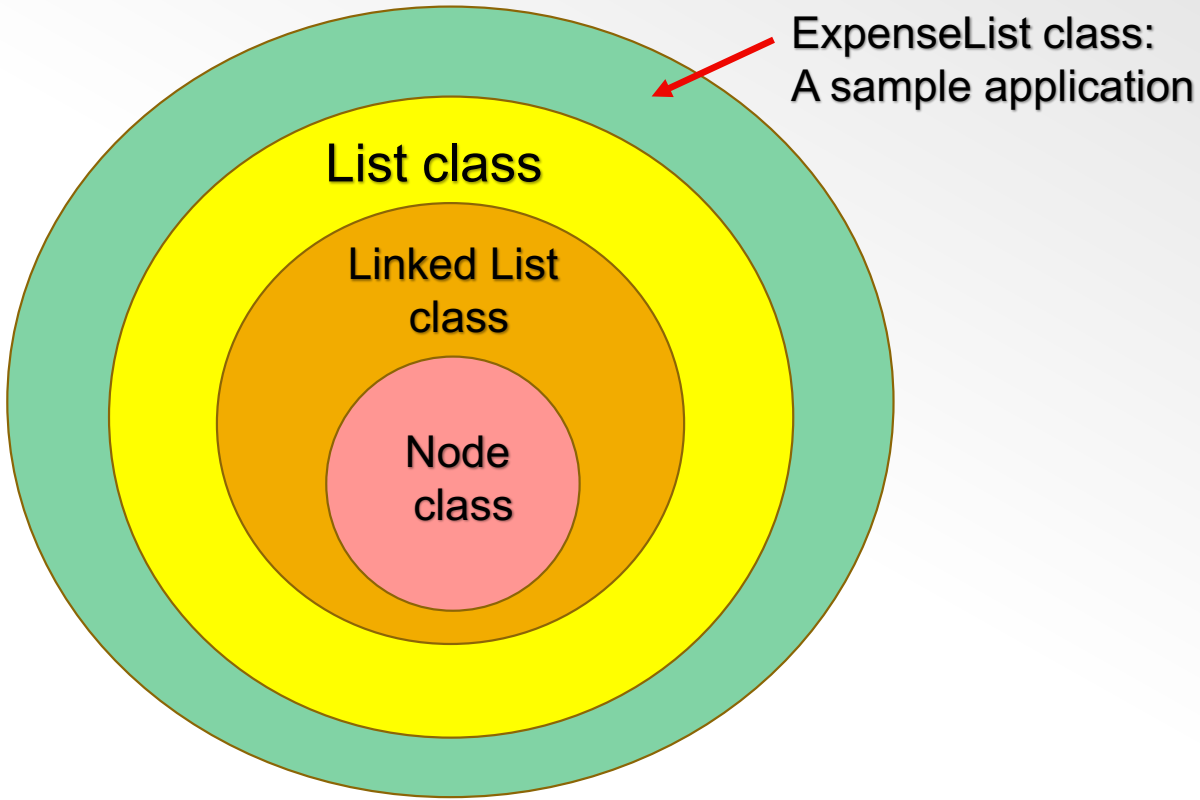
Similarly, an entry may be inserted anywhere in the list without
having to move other entries over to create space.


100
700
Raichu
450
450
Bulbasaur
700
700
Pikachu
600
600
MewTwo
null

A Linked List would be an excellent choice for the List class

- Memory is allocated/de-allocated on-demand.
- Items and can inserted or deleted without affecting other entries.
- One drawback is that it doesn't permit random access of an item like an array-type structure.
- But random access is typically not required in unordered lists.

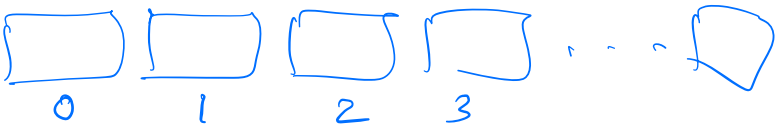
Here's our big picture for building the List class



Salient points about our implementation

- *We will build the classes as layers.*
- *Each layer sees only its immediate interface.*
- *We will use generics → the List class should work for any UnorderedList application.*
- *We will put a price tag (complexity) for each operation and achieve this complexity.*

A GENERIC UNORDERED LIST CLASS: List



Class List<T>

Constructor

List ()	**Creates an empty unordered list**

Methods

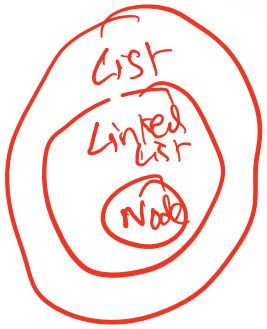
Name	What it does	Header	Price tag (complexity)
add	**Adds a given item to the list**	void add (T item)	O(1)
size	**Returns the number of items in the list**	int size ()	O(1)
isEmpty	**Returns true if the list is empty, false otherwise**	boolean isEmpty ()	O(1)
contains	**Returns the index of the first occurrence of an item (-1 if item not found)**	int contains (T item)	O(n)
clear	**Removes all items from the list**	void clear()	O(1)
remove	**Removes the first occurrence of the specified item from the list if it is present**	void remove (T item)	O(n)
removeAll	**Removes all occurrences of the specified item from the list (if present)**	void removeAll (T item)	O(n)
first	**Returns the reference to the first item in the list**	T first ()	O(1)
next	**Returns the reference to the next item in the list**	T next()	O(1)
enumerate	**Steps through the list and displays the items**	void enumerate()	O(n)

add to front →

keep a count →

Price tag constraints:

1. The implementation should be able to access the first item in the list in O(1) time, so that the add method can be implemented in O(1) time.
2. The implementation should maintain a count of the number of items in the list. The size method can then simply return this count in O(1) time. Furthermore, the isEmpty method can be implemented in O(1) time.
3. The implementation should be able to empty the list in O(1) time, that is, the clear method should run in O(1) time.
4. We will also use a cursor (pointer) to point to a current node on the list. This will help us to implement the first and next methods in O(1) time.
5. All other methods require sequential search and hence we have a price tag of O(n) time.



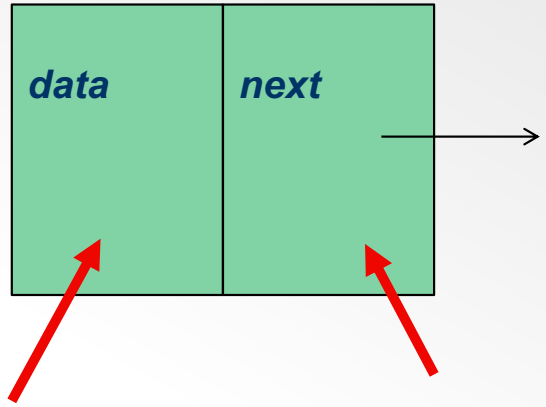
The Node Class: `public class Node<T>`

Attributes

- ***T data***
- ***Node<T> next***

Methods

- ***Constructor***
- ***get and set methods***



of type T

of type Node<T>

(It is a reference to Node<T>)

```
//Generic Node class
public class Node<T>{
        //instance variables
        private T data;
        private Node<T> next;

        //constructor
        public Node(T data, Node<T> next){
                this.data = data;
                this.next = next;
        }
        //getters
        public T getData(){
            return data;
        }
        public Node<T> getNext(){
            return next;
        }

        //setters
        public void setData(T data){
            this.data = data;
        }
        public void setNext(Node<T> next){
            this.next = next;
        }
}
```

Notes:

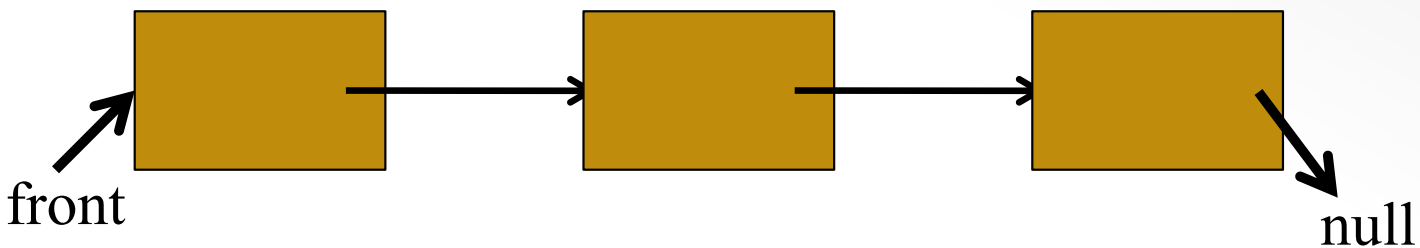
1. Put Node<T> in the class header.
2. Do not put Node<T> in the constructor.
3. Put Node<T> for all other references of the node variable.

The Linked List Class :
`public class LinkedList<T>`

Attributes

Node<T> front

int count

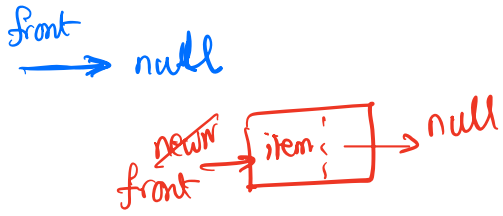


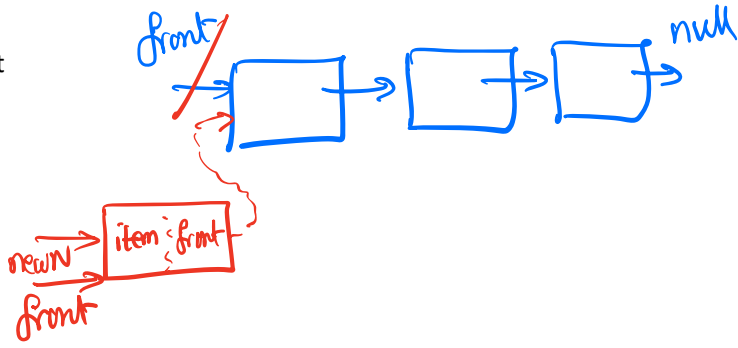
Methods in the Linked List class

Methods

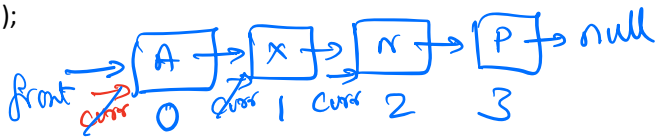
1. ***Constructor***
2. ***Add a node to the front of the linked list***
3. ***Get the size of the linked list***
4. ***Check if the linked list is empty***
5. ***Enumerate the linked list***
6. ***Get the item in a node given an index***
7. ***Insert a new node at a given index***
8. ***Set the node at a given index***
9. ***Find the index of the first occurrence of a given item*** → Search
10. ***Remove the node at a given index***
11. ***Remove the node of the first occurrence of a given item***
12. ***Remove all the nodes of a given item***

```
//Generic LinkedList class
//Builds upon the generic Node class
public class LinkedList<T>{
        //instance variables
        private Node<T> front;
        private int count;

        //constructor - creates an empty linked list
        public LinkedList(){
                front=null;
                count=0;
        }

        //method1 - add an item to the front of the linked list
        public void add(T item){
                Node<T> newN = new Node<T>(item, front);
                front = newN;
                count++;
        }

        //method2 - return the size of the linked list
        public int size(){return count;}

        //method3 - check if the linked list is empty
        public boolean isEmpty(){return (count==0);}

        //method4 - scan the list and display the items
        public void enumerate(){
                Node<T> curr = front;
                while (curr!=null){
                        System.out.print(curr.getData() + "-->");
                        curr = curr.getNext();
                }
                System.out.println();
        }

        //method5 - get the item at a given index
        public T getAt(int index){
                Node<T> curr = front;
                if (index<0 || index>=count){
                        System.out.println("Index out of bounds");
                        return null;
                }

                else{
                        for(int i=0;i<index;i++)
                                curr=curr.getNext();
                }
                return curr.getData();
        }
```





Works on an empty linked list as well!



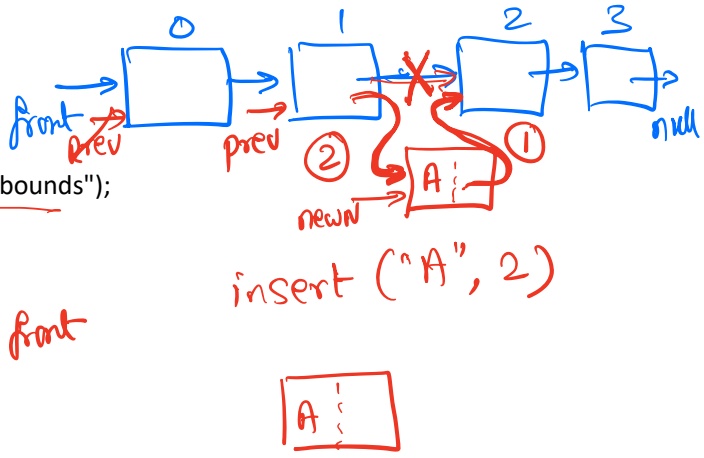
getAt(2) → N

index = 2

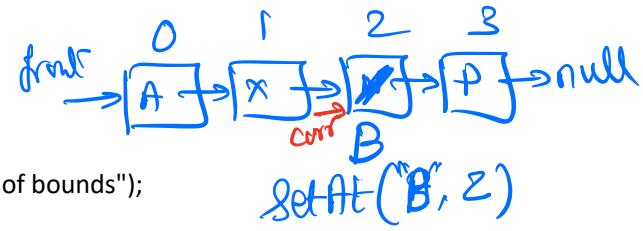
i=0 true
i=1 true
i=2 false

for (int i=0; i<index; i++) → get to the node at index

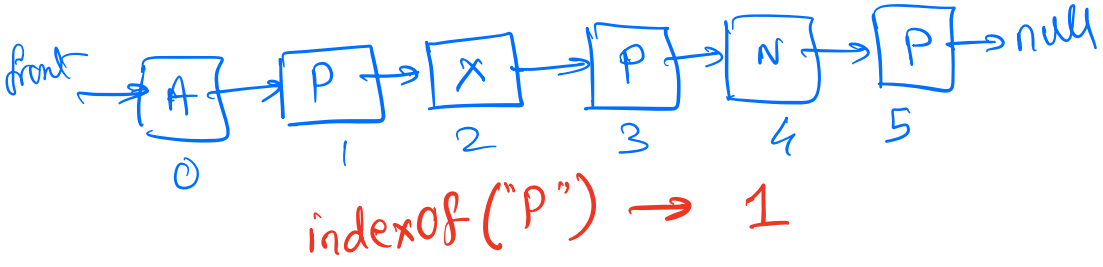
```
//method6 - insert an item at a given index
public void insertAt(T item, int index){
        if (index<0||index>count){
                System.out.println("Error. Index out of bounds");
                return;
        }
        else{
                if (index==0){
                        add(item);
                        return;
                }
                Node<T> prev=front;
                for(int i=0; i<index-1; i++)
                        prev=prev.getNext();
                Node<T> newN = new Node<T>(item, prev.getNext());
                prev.setNext(newN);
                count++;
        }
}
```



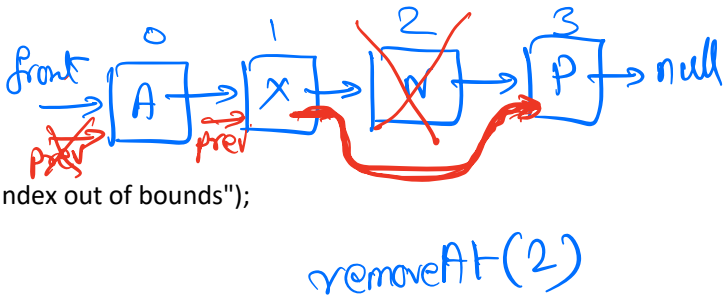

```
//method7 - set a node(given item and a given index)
public void setAt(T item, int index){
        if (index<0||index>=count){
                System.out.println("Can't set. Index out of bounds");
                return;
        }
        Node<T> curr = front;
        for(int i=0;i<index;i++)
                curr = curr.getNext();
        curr.setData(item);
}
```




```
//method8 - SEARCH METHOD: return the index of the first occurrence of a given item
//if item is not found, return -1
public int indexOf(T item){
        Node<T> curr = front;
        for(int i=0; i<count; i++){
                if (item.equals(curr.getData()))
                        return i; //item found; break out of the loop
                curr = curr.getNext();
        }
        return -1;
}
```



```
//method9 - remove an item at a given index
public T removeAt(int index){
        T result = null;
        if (index<0||index>=count){
                System.out.println("Can't remove. Index out of bounds");
                return null;
        }
        if (index==0) //first node has to be removed
        {
                result = front.getData();
                front = front.getNext();
        }
        else //not the first node
        {
                Node<T> prev = front;
                for (int i=0; i<index-1; i++)
                        prev = prev.getNext();
                result = prev.getNext().getData();
                prev.setNext(prev.getNext().getNext());
        }
        count--;
        return result;
}
```



removeAt(2)

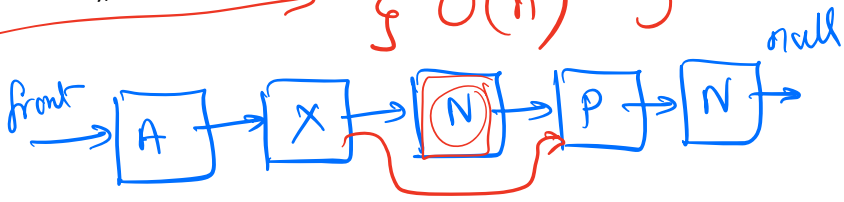
bypass

```
//method10 - remove the node of the first occurrence of a given item
public void remove(T item){
        int i = indexOf(item);
        if (i==-1)
                System.out.println("No such item");
        else
                T result = removeAt(i);
}
```

search for that item { O(n) } O(n)

{ O(n) }

to keep the compiler happy



remove ("N")

```
//method11 - remove all occurrences of a given item
public void removeAll(T item){
        Node<T> curr = front, prev = null;
        while (curr!=null){
                if (front.getData().equals(item)){ //front node has item
                        curr = curr.getNext();
                        front=front.getNext();
                        count--;
                }
                else if (curr==front){ //front node does not have item
                                        //but curr is at front
                        prev=curr;
                        curr=curr.getNext();
                }
                else //currr is somewhere in the middle
                {
                        if (curr!=null) //make sure curr is not null
                        {
                                if(curr.getData().equals(item)){
                                        prev.setNext(curr.getNext());
                                        curr=prev.getNext();
                                        count--;
                                }
                                else{
                                        prev=curr;
                                        curr=curr.getNext();
                                }
                        }
                }
        } //end while
} //end method11

} //end LinkedList class
```

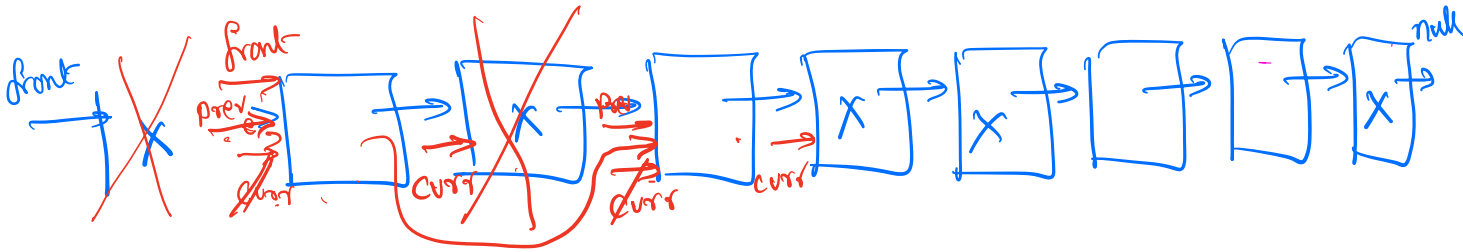
one big loop

Case 1

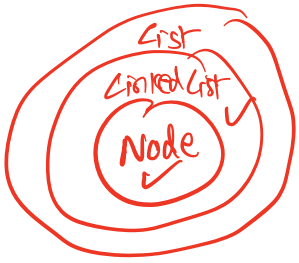
Case 2

could be redundant

Case 3



removeAll("x");



IMPLEMENTATION OF LIST CLASS: List<T>

The List class is built on top of the LinkedList class. It has two attributes – a LinkedList<T> and a pointer (cursor) to help step through the list. Notice how easy it is to implement the List class with the LinkedList methods.


```
public class List<T>
{
        private LinkedList<T> elements;
        private int cursor;

        public List()
        {
            elements = new LinkedList<T>();
            cursor = -1;
        }

        public void add(T item)
        {
            elements.add(item);


        }
        public int size()
        {
            return elements.size();

        }
        public boolean isEmpty()
        {
            return elements.isEmpty();

        }
        public boolean contains(T item)
        {
            return (elements.indexOf(item) != -1);

        }
        public void remove(T item)
        {
            elements.remove(item);

        }
```

```
public void removeAll(T item)
{
        elements.removeAll(item);
}
public void clear()
{
        elements.clear();
}
public void enumerate()
{
        elements.enumerate();
}
public T first()
{
    if (elements.size() == 0) return null;
    cursor = 0;
    return elements.getAt(cursor);
}
public T next()
{
    if (cursor < 0 || cursor == elements.size() - 1)
        return null;
    cursor++;
    return elements.getAt(cursor);
}

}
```

will be used together to step thru' the list.

EXAMPLE APPLICATION ON LIST CLASS – EXPENSE LIST

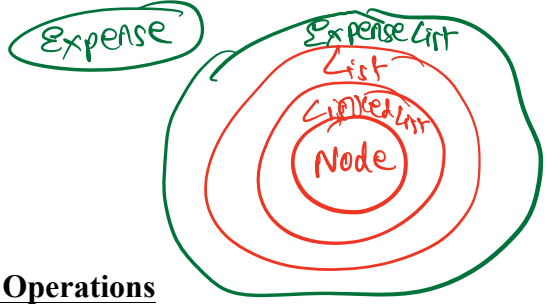
Now we will build a sample application using the List class. We need to store a list of expenses of the following form:

Date	Description	Amount
Aug 24, 2024	Book	45.00
Sep. 4, 2024	Groceries	149.99
Aug 24, 2024	Gas	88.89
…		

We first build an Expense class to model each entry (Date, Description, Amount) and then use this to build an ExpenseList class.

Expense Class

```
import java.text.DecimalFormat;
public class Expense
{
	private String date;
	private String desc;
	private double amount;
	private static final DecimalFormat dollar = new DecimalFormat("0.00");

	public Expense(String date, String desc, double amount)
	{
		this.date = date;
		this.desc = desc;
		this.amount = amount;
	}
	public String toString()
	{
		return date + "\t" + desc + "\t\t" + dollar.format(amount) + "\n";
	}
	public boolean equals(Expense other)
	{
		return
(desc.equals(other.getDesc())&&date.equals(other.getDate())&&amount==other.getAmo
unt());

	}
	public String getDesc()
	{
		return desc;
	}
	public String getDate()
	{
		return date;
	}
	public double getAmount()
	{
		return amount;
	}

}
```



ExpenseList Class

Attributes	Operations
A list (of expenses)	Constructor to create an empty list add an expense to the list check if the list is empty check if the list contains an expense get the first item get the next item enumerate find the maximum expense amount find the minimum expense amount find the average expense amount find the total amount of expenses find the total spent on a given item

no more generics!

```
public class ExpenseList
{
    private List<Expense> expenses;

    public ExpenseList()
    {
        expenses = new List<Expense>();
    }
    public void add(Expense exp)
    {
        expenses.add(exp);
    }

    public boolean isEmpty()
    {
        return expenses.isEmpty();
    }
    public boolean contains(Expense exp)
    {
        return expenses.contains(exp);
    }
    public Expense first()
    {
        return expenses.first();
    }
```

```
public Expense next()
{
        return expenses.next();
}
public void enumerate()
{
        expenses.enumerate();
}

public double maxExpense()
{
        double max = 0.0;
        Expense exp = expenses.first();
        while (exp != null)
        {   if (exp.getAmount() > max)
                max = exp.getAmount();
            exp = expenses.next();
        }
        return max;
}
public double minExpense()
{
    if (expenses.isEmpty()) return 0;
    Expense exp = expenses.first();
    double min = exp.getAmount();
    while (exp != null)
    {   if (exp.getAmount() < min)
            min = exp.getAmount();
        exp = expenses.next();
    }
    return min;
}
```

```
    public double avgExpense()
    {

    }
    public double totalExpense()
    {

    }
    public double amountSpentOn(String expItem)
    {

    }

}
```

Demo program

The following program reads a text file with expense items, creates an unordered list, prints the list and also some expense statistics. It is assumed that the input file lists the date, description, expense one on each separate line as follows:

```
Aug 24, 2024
Book
45.00
Sep 4, 2024
Groceries 149.99
Aug 24, 2024
Gas
88.89
```

```
import java.util.Scanner;
import java.io.*;
public class ExpenseListDemo
{
	public static void main(String[] args) throws IOException
	{
		Scanner keyboard = new Scanner(System.in);
		System.out.print("Enter the filename to read from: ");
		String filename = keyboard.nextLine();

		File file = new File(filename);
		Scanner inputFile = new Scanner(file);

		ExpenseList expenses = new ExpenseList();
		String date, desc, cost;
		Expense exp=null;

		while (inputFile.hasNext())
		{
			date = inputFile.nextLine();
			desc = inputFile.nextLine();
			cost = inputFile.nextLine();
			exp = new Expense(date, desc, Double.parseDouble(cost));
			expenses.add(exp);
		}

		inputFile.close();

		System.out.println("Expenses");
		expenses.enumerate();
		System.out.println("The max expense was: " +
expenses.maxExpense());
		System.out.println("The min expense was: " +
expenses.minExpense());
		System.out.println("The avg expense was: " +
expenses.avgExpense());
		System.out.println("The amount spent on groceries was: "
+ expenses.amountSpentOn("Groceries"));
		System.out.println("The total expenses were: " +
expenses.totalExpense());
	}
}
```

COMPLEXITY ANALYSIS *(Did we meet the price tag expectations?)*

Operation	Complexity
add	O(1) → add to front
size	O(1) → count
isEmpty	O(1)
remove	O(n)
removeAll	O(n)
contains	O(n)
clear	O(1)
first	O(1)
next	O(n) → uses getAt

COMPLEXITY ANALYSIS OF SEARCH

Best Case Complexity

comparisons in the best case = 1 O(1)

Worst Case Complexity

Successful search # comparisons = n

Unsuccessful search # comparisons = n

} O(n)

Average Case Complexity

In general, if it takes C_i comparisons for a successful search at the ith entry, then the average number of comparisons is:

$$\frac{C_1 + C_2 + C_3 + \cdots + C_n}{n}$$

In an unordered list, it takes 1 comparison for a successful search at the first entry, 2 comparisons for a successful search at the second entry and so on. Hence the above formula becomes:

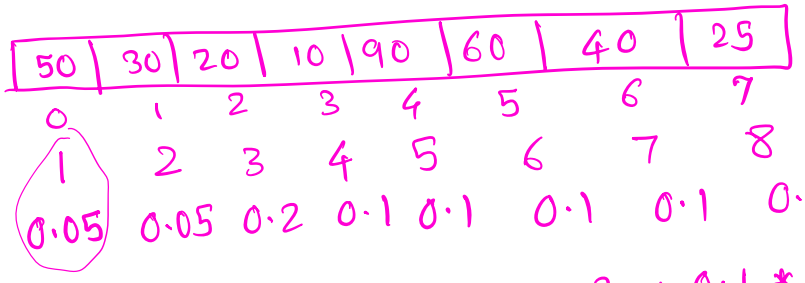
$$\frac{1 + 2 + 3 + \cdots + n}{n} = \frac{\frac{n(n+1)}{2}}{n}$$

$$= \frac{n+1}{2} \rightarrow O(n)$$

But an assumption that we made is that the probability of searching for all items is the same, that is, any item is searched with the same likelihood as any other. Suppose this is not true. Let Pi be the likelihood of searching the ith item, then the average number of comparisons is:

P1 * C1 + P2 * C2 + P3 * C3 + ... + Pn * Cn

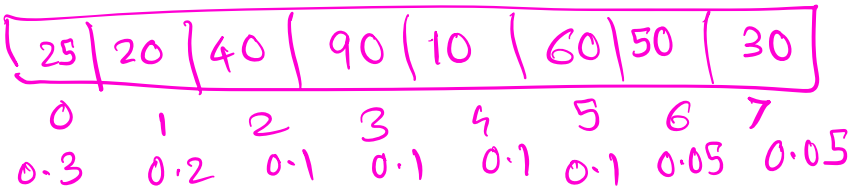
Example:



Then average # comparisons = 0.05*1 + 0.05*2 + 0.2*3 + 0.1*4 + 0.1*5 + 0.1*6 + 0.1*7 + 0.3*8 = 5.35

Note that if we assume equal search probabilities, P_i would be 1/n. For sequential search on an unordered list, we would then arrive at (n + 1)/2 for the average number of comparisons, which tallies with our earlier result.

Order them in decreasing order of probs



Avg. # comparisons: = 3.25

A smart technique to improve average case complexity

- Each time an item x is searched, move it to the front of the list.
- The more number of times an item is search, the more towards the front of the list it will be.