Data structures II – Linked Lists, Stacks and HashTables

LINKED LISTS (contd.)

From Last Week

1. Linked Lists
2. Operations available
   - `public boolean isEmpty();`
   - `public void addFirst(String element);`
   - `public void removeFirst();`
   - `public void addLast(String element);`
   - `public void removeLast();`
   - `public void addMid(String element, String entryafter);`
   - `public void removeMid(String element);`
   - `public static void printList(SLinkedList thelist)`

Inserting at the Tail

1. Allocate a new node

   `new StringNode();`

2. Insert new element

   `new StringNode(element,);`

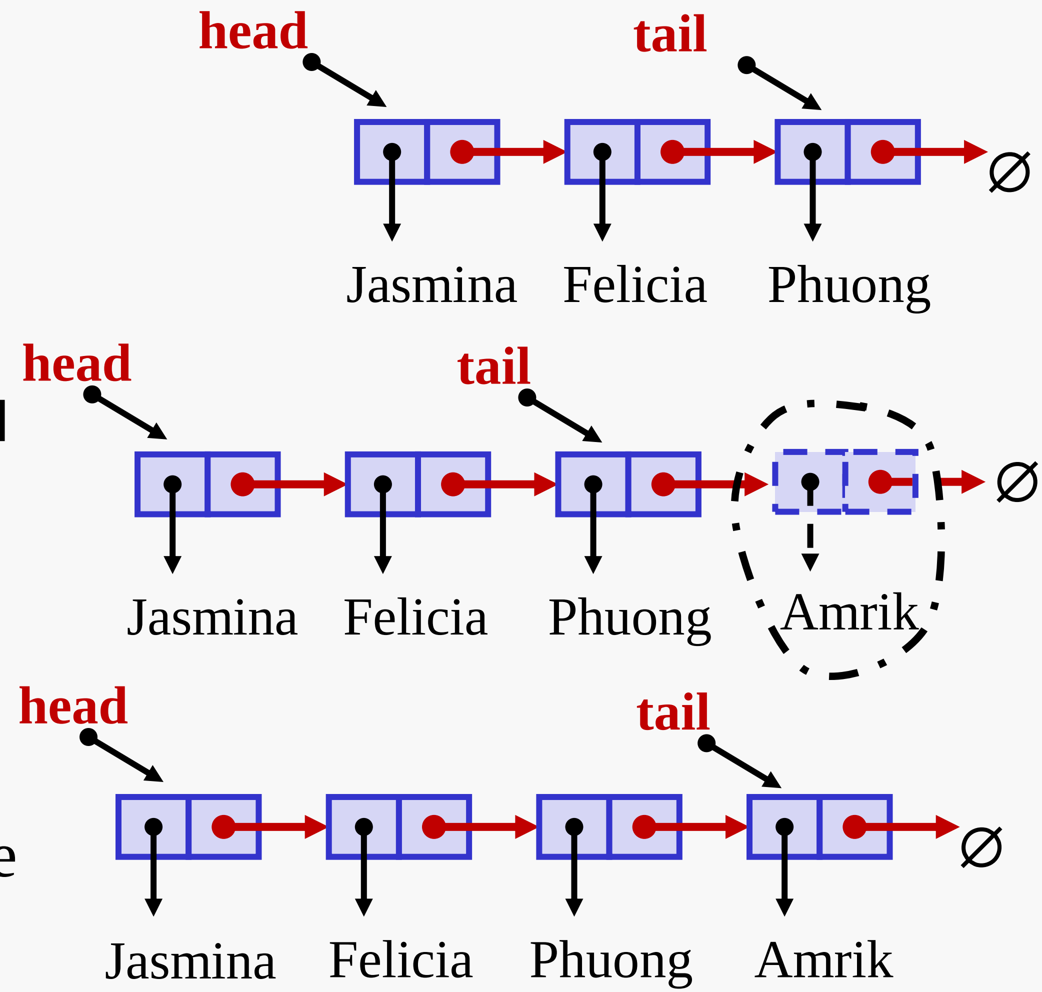
3. Have new node point to `null`

   `new StringNode(element,null);`

4. Have old last node point to new node

   `tail.setNext( new StringNode(element,null));`

5. `Tail` now points to new node



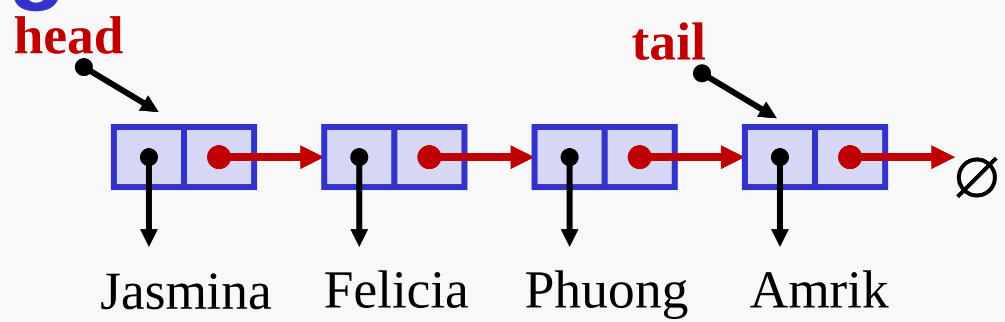
Inserting at the Tail

- The following method inserts a new node at the `tail` of the list.

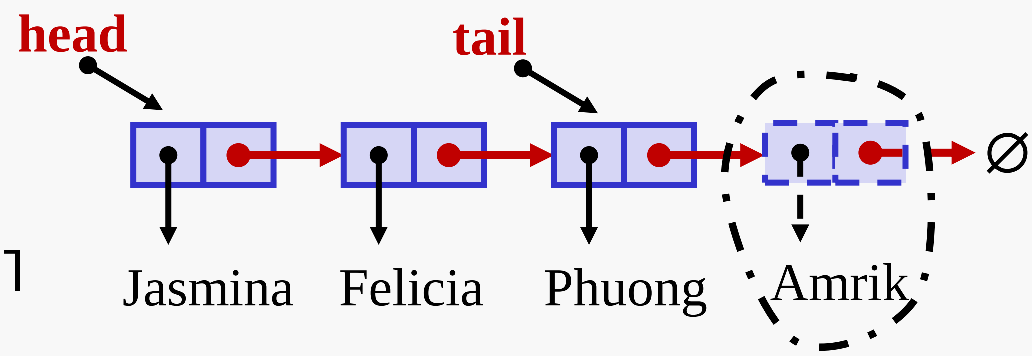
```
private void addLast(String element) {
    StringNode tail;
    tail = head;
    while (tail.getNext() != null) {
        tail = tail.getNext();
        }
    //insert new node at end of list
    tail.setNext( new StringNode(element,null));
}
```

- the statement **`tail.setNext( new StringNode(element,null))`** does the steps
  1. Allocate a new node
  2. Insert new element
  3. Have new node point to `null`
  4. Have old last node point to new node

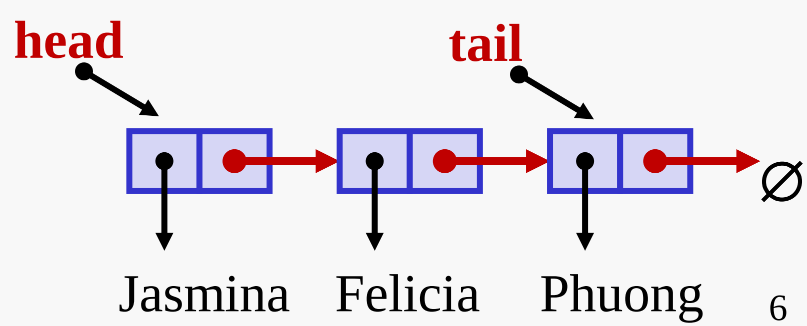
Removing at the Tail



1. Set up `tail` to point to previous node in the list



2. Set `next` node of new `tail` to be `null`
3. Allow garbage collector to reclaim the former last node



Removing at the Tail

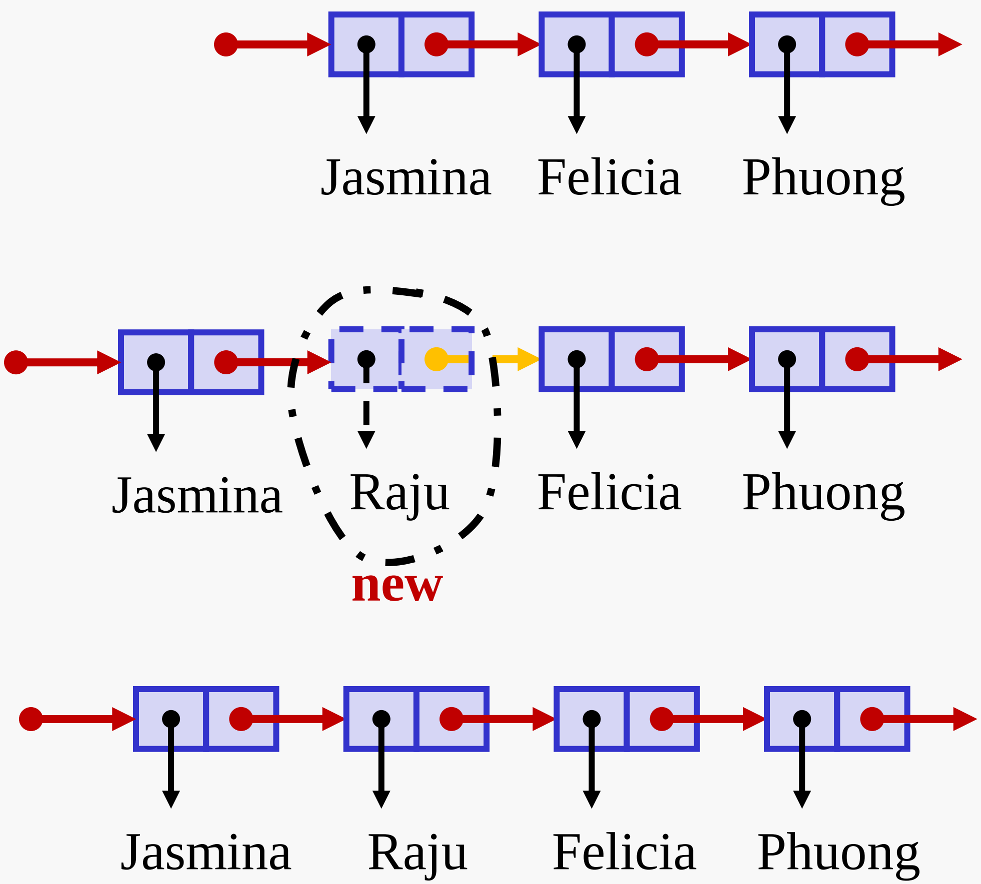
- The following method removes the node at the `tail` of the list. The node that pointed to the old tail now becomes the new `tail` of the list.

```
private void removeLast() {
    StringNode temp, previous;
    temp = head;
    previous = temp;
    //go to last node and remember previous node at all times
    while (temp != null && temp.getNext() != null) {
        previous = temp;
        temp = temp.getNext();
        }
    if (previous != null) {
       //remove last node
       previous.setNext(null);
       }
    else {
       throw new NoSuchElementException();
       }
}
```

1. Set up `tail` to point to previous node in the list
2. Set `next` node of new `tail` to be `null`
3. Allow garbage collector to reclaim the former last node

Inserting inside the List

1. Allocate a new node
2. Insert new element
3. Insert new node after prescribed node
4. Have new node point to next node of prescribed



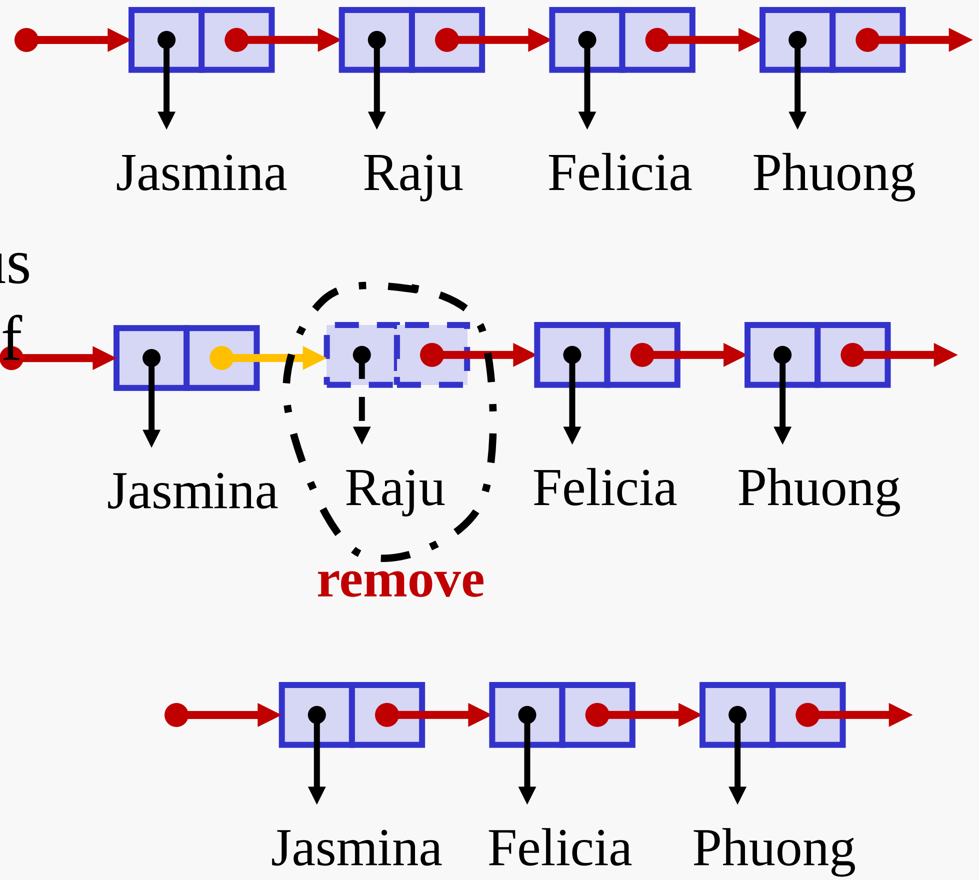
Inserting inside the List

- The following method inserts a new node inside an existing list after a prescribed element.

```
private void addMid(String element, String entryafter) {
    StringNode curnode;
    curnode = head;
    //find node identified by entryafter
    while (curnode != null && curnode.getElement() != entryafter) {
        curnode = curnode.getNext();
        }
    //if first occurrence of element entryafter was located then insert new node
    if (curnode != null) {
        StringNode newnode = new StringNode(element,curnode.getNext());
        curnode.setNext(newnode);
        }
}
```

- the statement **`new StringNode(element,curnode.getNext())`** does the steps
  1. Allocate a new node
  2. Insert new element
  3. Have new node point to next node of prescribed (**`curnode.getNext()`**)
- the assignment **`curnode.setNext(newnode)`** does the last step
  4. Insert new node after prescribed node

Removing inside the List



1. Set the `next` field of previous node to point to `next` field of discarded node
2. Set `next` node of discarded node to be `null`
3. Allow garbage collector to reclaim the discarded node

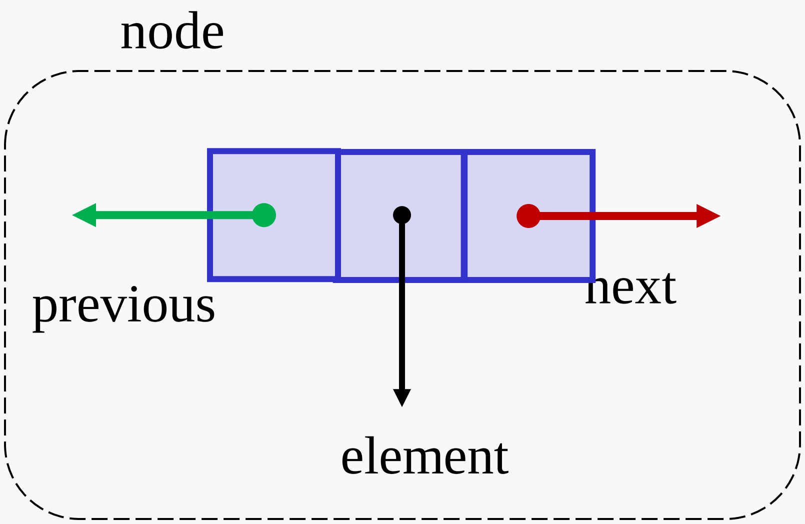
Removing inside the List

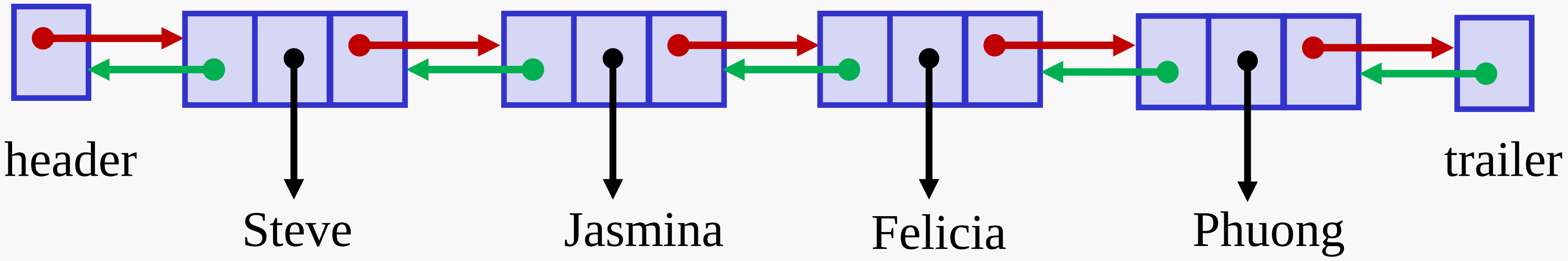
- The following method removes a node inside an existing list after a prescribed node.

```
private void removeMid(int element) {
   StringNode temp, previous;
   temp = head.getNext();
   previous = null;
   //go to node containing element and remember previous node at all times
   while (temp.getElement() != element && temp.getNext() != null) {
       previous = temp;
       temp = temp.getNext();
       }
   if (previous != null && temp.getNext() != null) {
       //not first or last node so we can remove node defined by temp.
       // set the previous node to that after temp
       previous.setNext(temp.getNext());
       temp.setNext(null);
       }
   else {
       throw new NoSuchElementException();
       }
}
```

Doubly Linked List

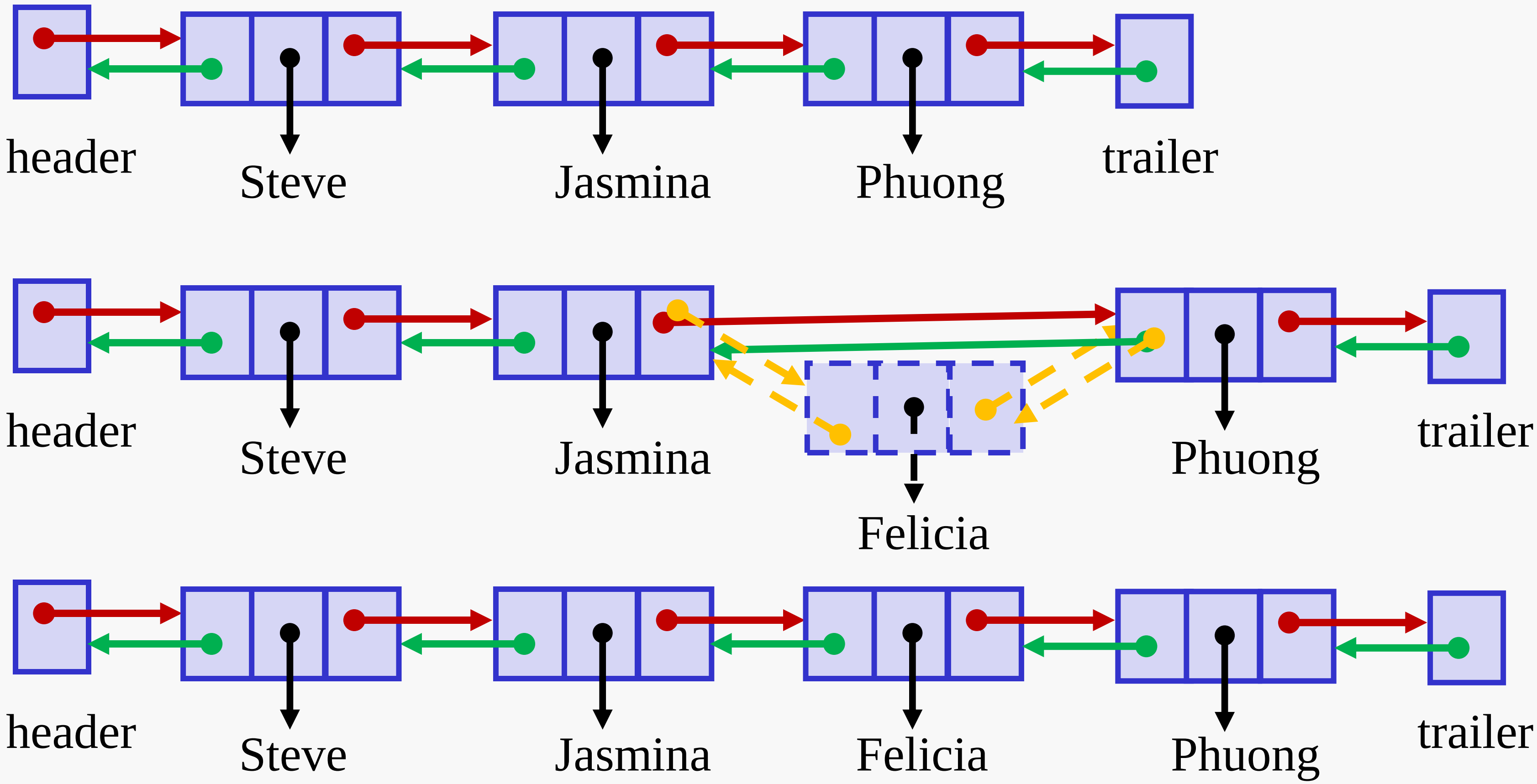
- A doubly linked list is often more convenient
- Each node stores
  - element
  - link to the previous node
  - link to the next node
- Special header and trailer nodes





Insertion for Doubly Linked List

- Process of `insertAfter(node(Jasmina),Felicia)`, which does the following

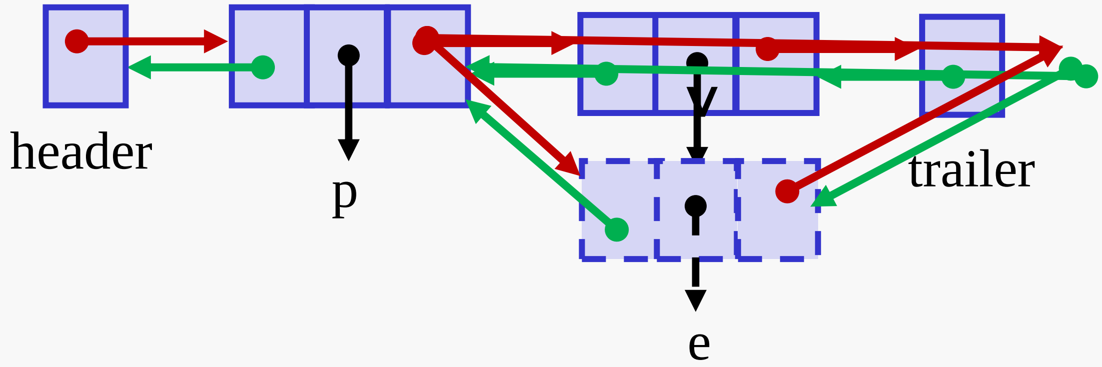


Insertion Algorithm for Doubly Linked List

Algorithm `insertAfter(node(p),e):`

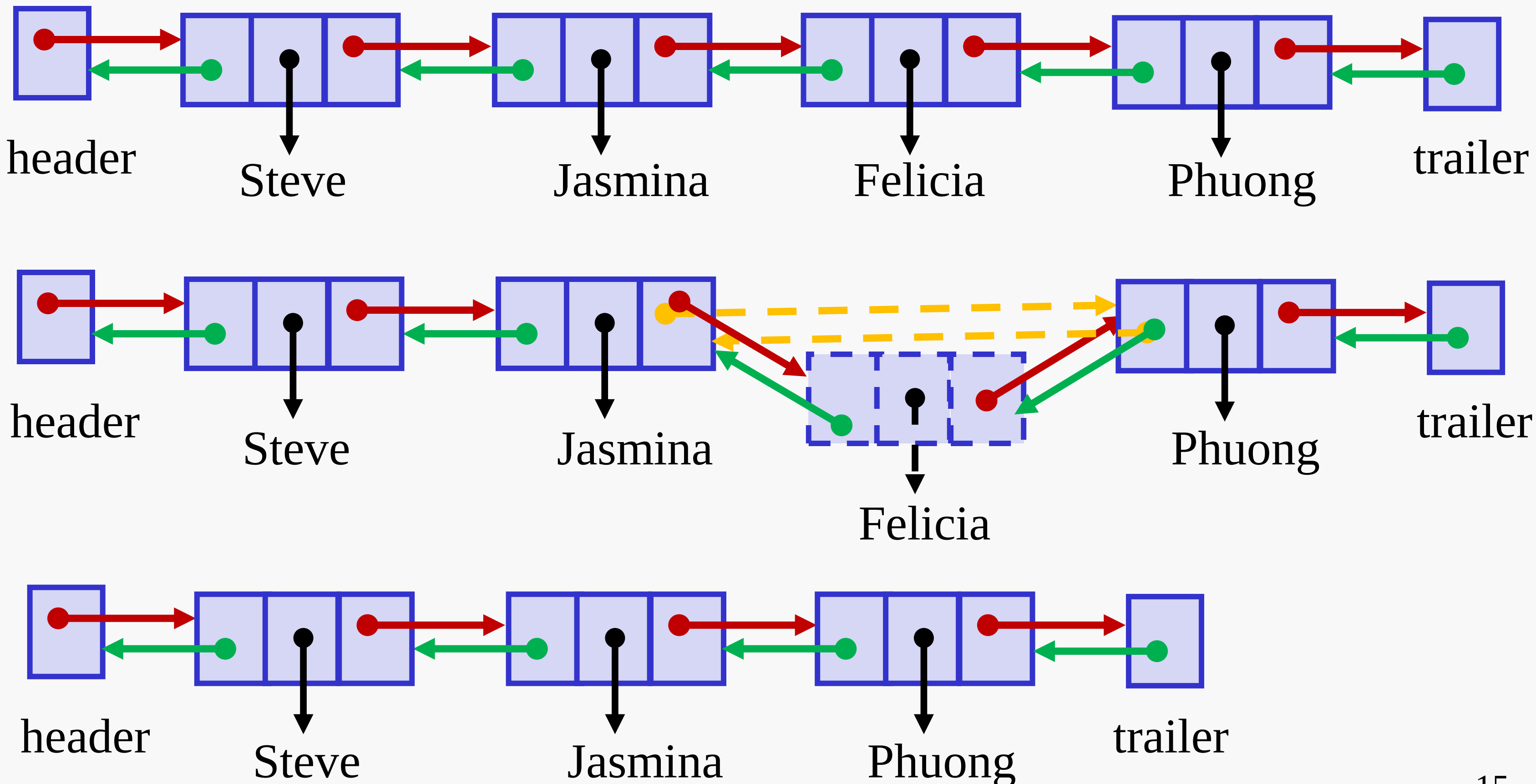
Create a new node `v`

`v.setElement(e)`	{set object `e` as the new element}
`v.setPrev(p)`	{link `v` to its predecessor}
`v.setNext(p.getNext())`	{link `v` to its successor}
`(p.getNext()).setPrev(v)`	{link `p`'s old successor to `v`}
`p.setNext(v)`	{link `p` to its new successor, `v`}
return `v`	{the position for the element `e`}



Deletion for Doubly Linked List

- Process of `remove(node(Felicia))`, which does the following



Deletion Algorithm for Doubly Linked List

Algorithm `remove(node(p),p):`

```
(p.getPrev()).setNext(p.getNext())
```

{link predecessor of p to its successor}

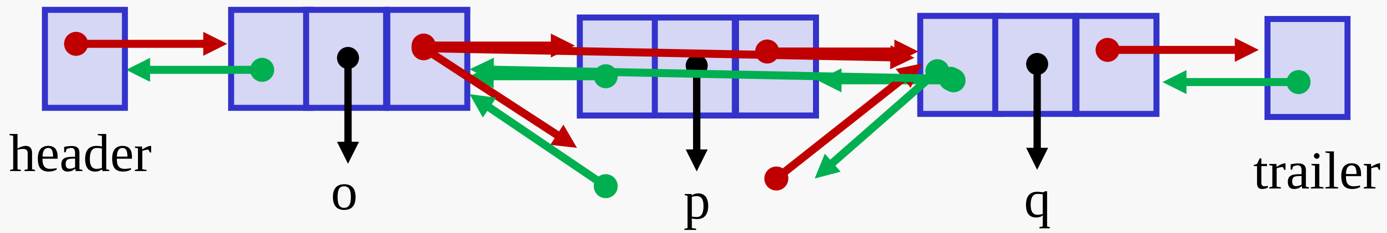
```
(p.getNext()).setPrev(p.getPrev())
```

{link successor of p to its predecessor}

`p.setPrev(null)` {invalidate predeccessor of p}

`p.setNext(null)` {invalidate successor of p}

return



STACKS

What is a stack

- The usual analogy is the “stack of plates”
- A way of buffering a stream of objects, in which the the **L**ast **I**n is the **F**irst **O**ut (LIFO)
- What might we use a stack for?
- Functional requirements of a stack
  - Two basic methods, ***push*** and ***pop***.
  - The first plate begins the pile, the next is placed on top of the first and so on.
  - A plate may be removed from the pile at any time, but only from the top.
  - ***Push***ing a plate onto the pile increases the number on the pile (by 1)
  - ***Pop***ping a plate from the pile decreases the number on the pile (by 1)

Calculator example

- Say we want to build a calculator to evaluate

```
(3 + (2 * 5)) / 6
```

- Work your way from the outside of the expression to the inside putting the operands (numbers and mathematical operators) onto the stack

```
2
*
5
+
3
/
6
```

The stack has 7 elements on it and `2` is at the top of the stack.

To evaluate, put the item at the head into a variable and **pop** each element in turn performing the specified calculation on the variable.

Finished when the stack is empty.

Simple stack implementation

- Stack is represented by a `private` array of objects …

```
public class SimpleArrayStackofchars implements Stack {
  protected int capacity;       // The actual capacity of the stack array
  public static final int CAPACITY = 2; // default array capacity
  protected Object S[];         // Generic array used to implement the stack
  protected int top = -1;       // index for the top of the stack
                                // if top is -1 -> empty stack
  public SimpleArrayStackofchars() {
    this(CAPACITY); // default capacity
  }
  public SimpleArrayStackofchars(int cap) {
    capacity = cap;
    S = new Object[capacity];
  }
```

Simple stack implementation contd

```
public int size() {
  return (top + 1);
}
public boolean isEmpty() {
  return (top == -1);
}
public void push(Object element) throws FullStackException {
  if (size() == capacity) {
    throw new FullStackException("Stack is full. Stack size max is
 "+ capacity);
    // can replace previous line with code to double stack size
    // doubleArray();
  }
  S[++top] = element;
}
public Object top() throws EmptyStackException {
  if (isEmpty())
    throw new EmptyStackException("Stack is empty.");
  return S[top];
}
```

Simple stack implementation contd

```
public Object pop() throws EmptyStackException {
    Object element;
    if (isEmpty())
      throw new EmptyStackException("Stack is empty.");
    element = S[top];
    S[top--] = null; // dereference S[top] for garbage collection.
    return element;
    }
}
```

Interface for Stack

- Note: clients of the **`Stack`** class only have access to the stack through the public methods **`isEmpty()`**, **`push()`** and **`pop()`**.
- The actual stack array and the pointer **`top`** are `private` and cannot be directly manipulated by the client.
- the Java construct of the `Stack interface` is:

```
public interface Stack
  {
  public boolean isEmpty();
  public void push(Object items);
  public Object pop();
  }
```

Implementation of Stack

- Create `doubleArray` method so **`SimpleArrayStackofchars`** will be able to deal with the situation when a stack needs to be re-sized because it is full.

```
private void doubleArray( ) {
  Object [ ] newArray;
  System.out.println("Stack is full (max size was "+capacity+").
                    Increasing to "+(2*capacity));
  //double variable capacity
  capacity = 2*capacity;
  newArray = new Object[ capacity ];
  for( int i = 0; i < S.length; i++ )
      newArray[ i ] = S[ i ];
  S = newArray;
}
```

Modify `push` method also...

```
public void push(Object element) {
// replaced throw exception if stack is full with code to double stack size
if (size() == capacity) doubleArray();
S[++top] = element;
}
```

Demo use of SimpleArrayStackofchars

Can't instantiate `Stack` directly, it's only an **interface.** Need to "object"-ify items to put them on stack

```
  public static void main(String[] args) {
   Stack S = new SimpleArrayStackofchars();
   S.push(1);
   S.push(2);
   S.push(3);
   S.push(4);
   S.push(5);
   S.push(6);
   while (!S.isEmpty()) {
     System.out.println(S.pop());
   }
}
```

With a default capacity size of 2, executing the above main will produce the output...

```
Stack is full (max size was 2). Increasing to 4  <- when attempting to push 3
Stack is full (max size was 4). Increasing to 8  <- when attempting to push 5
6
5
4
3
2
1
```

Linked list implementation

- We used arrays to demonstrate stacks so we relied on statically declared array size and techniques to copy arrays to new larger arrays if required
- This is not optimally efficient although it works just fine!
- It would be more efficient to use a data structure that easily and dynamically adjusted its size such that it provided precisely the right amount of data storage for the task at any time (linked lists).

HASH TABLES

Maps

- A ***map*** models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items (seen with linked lists)
- Multiple entries with the same key are **not allowed**
- Applications:
  - address book
  - student-record database
  - Compilers
  - Browser caches

The Map ADT

- Some map ADT methods:
  - `get(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return `null`

    e.g. `get(Alison)` returns `02085551234`
  - `put(k,v)`: insert entry `(k,v)` into the map `M`; if key `k` is not already in `M`, then return `null`; else, return old value associated with `k`

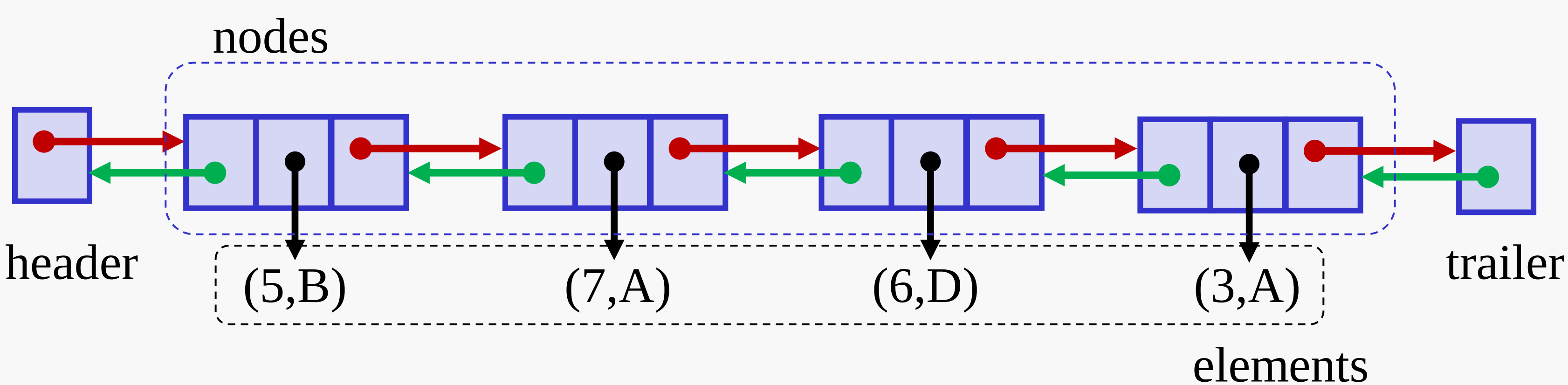
    e.g. `put(Joe,02085555678)` returns old tel no `01322400400`
  - `remove(k)`: if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return `null` e.g. `remove(Joe)` returns `02085555678`
  - `size()`, `isEmpty()`
  - `...`

Example use of a Map

Operation	*Output*	*Map*
`isEmpty()`	true	Ø
`put(5,A)`	null	**(5,A)**
`put(7,B)`	null	(5,A),**(7,B)**
`put(2,C)`	null	(5,A),(7,B),**(2,C)**
`put(8,D)`	null	(5,A),(7,B),(2,C),**(8,D)**
`put(2,E)`	**C**	(5,A),(7,B),(2,**E**),(8,D)
`get(7)`	**B**	(5,A),(7,**B**),(2,E),(8,D)
`get(4)`	**null**	(5,A),(7,B),(2,E),(8,D)
`get(2)`	**E**	(5,A),(7,B),(2,**E**),(8,D)
`size()`	**4**	(5,A),(7,B),(2,E),(8,D)
`remove(5)`	**A**	(7,B),(2,E),(8,D)
`remove(2)`	**E**	(7,B),(8,D)
`get(2)`	**null**	(7,B),(8,D)
`isEmpty()`	**false**	(7,B),(8,D)

A Simple List-based Map

- We can (inefficiently) implement a map using an unsorted list
  - We store the items of the map in a list `S` (based on a doubly-linked list), in arbitrary order



The `get(k)` and `put(k,v)` Algorithms

Algorithm `get(k)`:

set `temp to header`

scan through list looking for node `k` i.e. `temp=k`

if match is found return `value` otherwise return `null`

Algorithm `put(k,v)`:

set `temp to header`

scan through list looking for node `k` i.e. `temp=k`

if match is not found

Add `k` and `v` to the end of the list

Increment the total number of nodes counter (N)

return null

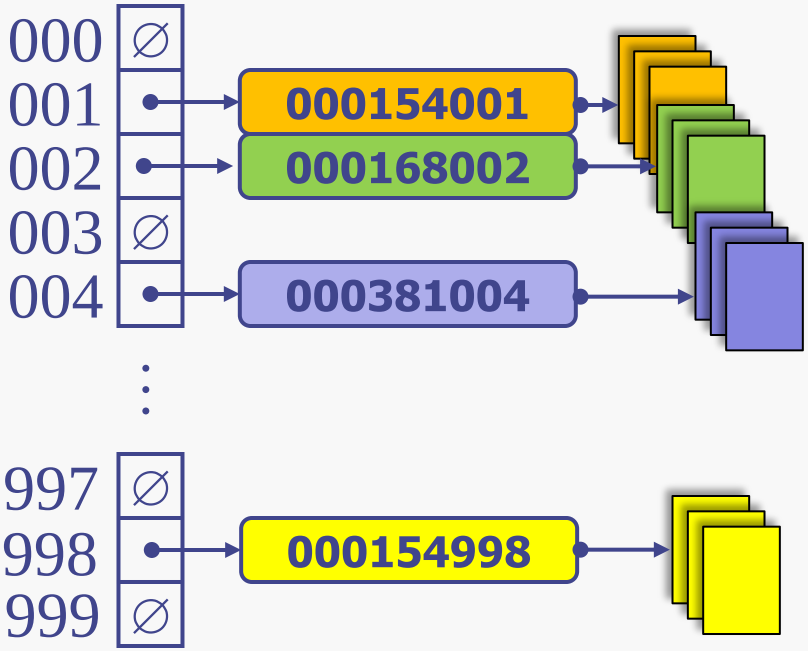
else

replace old value with `v`

return old value

Hash Tables

- Efficient storage and retrieval of information e.g. Obtaining student records based on banner id from a database



Hash Functions and Hash Tables

- A *hash function* `h` maps keys of a given type to integers in a fixed interval `[0,N-1]`
- Example:

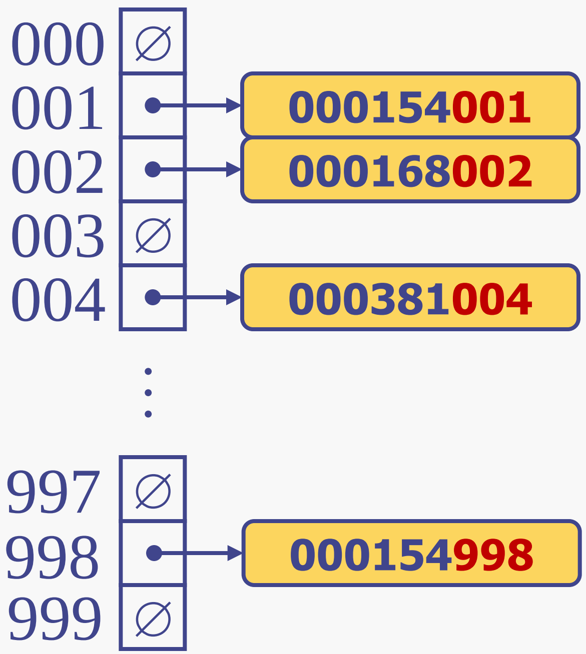
  `h(x) = x mod N`

  is a hash function for integer keys and ensures `h` lies between `0` to `N-1`
- The integer `h(x)` is called the hash value of key `x`
- A *hash table* for a given key type consists of
  - hash function `h`
  - array (called table) of size `N`
- When implementing a hash table, the goal is to store item `(k,i)` at index `i = h(k)` so that the items are dispersed
- **Duplicate key entries are not allowed**

Example hash table

- We design a hash table for a data structure storing entries as `(id, Name)`, where `id` (banner id) is a nine-digit positive integer

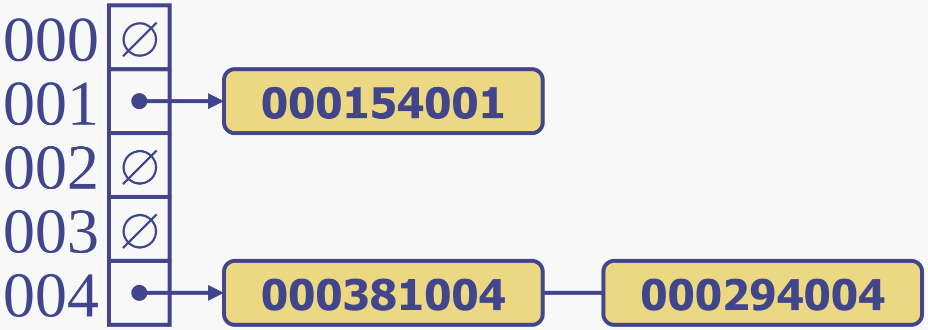
- Our hash table uses an array of size `N = 1000` and the hash function `h(x)` = last three digits of `x`



Collision Handling

- Collisions occur when different elements are mapped to the same cell

- **Solution 1 -** ***Separate Chaining***: let each cell in the table point to a linked list of entries that map there



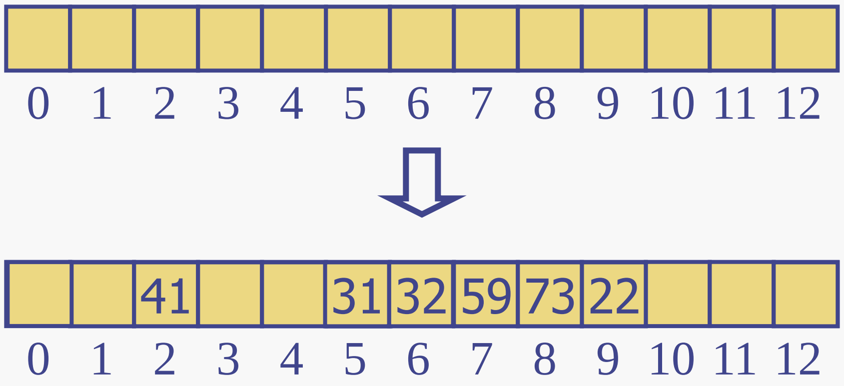
- Separate chaining is simple, but requires additional memory outside the table

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- **Solution 2 -** ***Linear probing*** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a “probe”
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Example:

- `h(x) = x mod 13`
- Insert keys `18,41,22,44, 59,32,31,73` in this order



Double Hashing

- **Solution 3 -** ***Double hashing*** uses a secondary hash function `d(k)` and handles collisions by placing an item in the first available cell of the series

  `(i + jd(k)) mod N`

  for j `= 0,1,…,N-1`
- `d(k)` also uses a mod function and relies on a selected prime number…
- Not covered in this course...☺

Acknowledgements

- Some of this material has been taken from a variety of sources
- the most notable of which is from

Goodrich and Tamassia "Data Structures and Algorithms in Java" John Wiley & Sons.