Tutorial 4

Set ADT and Data Structure using Linked Lists

Lecturer: Dr Andrew Hines *TA: Esri Ni*

4.1 Set ADT (recap from last week)

4.1.1 Definition

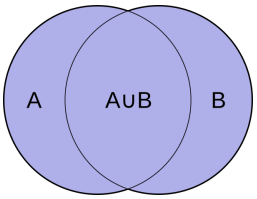
A set is an *unordered* collection of *distinct* objects.

- There are no duplicate elements in a set.
- There is no notion of a key or an order – elements are not organised in the set.

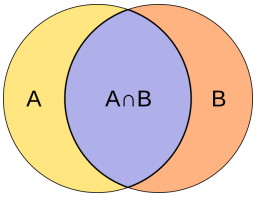
Unlike a sequence, we are concerned with the contents but not with the index of items, i.e. $S = \{1, 2, 3\}$ is equivalent to $S = \{2, 1, 3\}$. However we do not want duplicates, so $S = \{1, 2, 2\}$ would not be valid.
Sets are used a lot in data science/data analysis.

4.1.2 Set Operations

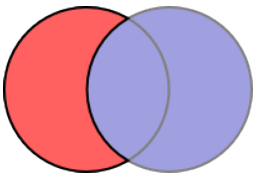
A reminder of set operations from your Venn diagrams math classes.
Union: Combine set A with the union of A and B, that is, $A \cup B$ is the combined (but without duplicate) elements of A and B.



Intersection: Intersection of A and B, that is, $A \cap B$ is the elements common to both sets.



Subtraction/Difference: Difference of A and B, that is, $A - B$, i.e. remove the elements common to both from A.



4.2 A Linked-list Based Set

Here is the pseudo-code for the `add` algorithm, implemented with a linked list, to get you started:

Algorithm 1 add

Input: L a linked-list representing a set, e an element (node)
Output: none - e is added to the set if not already present

```
p ← L
found ← false
while p is not null and found = false do
  if p.element() = e.element() then
    found ← true
  end if
  p ← p.next()
end while
if found = false then
  e.next() ← L.next()
  L ← e
end if
```

Walking through this, we first assign the address of our set L to the working pointer p. We set the found flag to false as we to begin with we assume the element to be added is not in the list unless proven otherwise. The while loop will keep going as long as p is not Null (i.e. unless we have not reached the terminator next element pointer at the end of the linked list) and that we have not found the element. If we find the element we set found to true, otherwise we move to the next item in the list by assigning the next address, $p.next()$ to p and iterate through the list.
If we get to the end of the list and the $found$ flag is still false then the element does not exist in the linked list and we can append it on to the beginning of the linked list. We do this by substituting the address of the beginning of the list into $e.next()$ and assigning e to be the address of the start of the linked list.

4.3 Exercises

Write the pseudo-code algorithms of the operations on the set ADT: `remove`, `size`, `is_empty`, `union`, `intersection`, `difference`, assuming the set ADT is implemented using a Linked-list.
Hint: for remove, you may want to access the address of the node after the next node, e.g. p.next().next().
Don't worry if you cannot work all of these out. The learning is in puzzling through them and thinking about how linked list implementations operate and how to break the problem down (e.g. using the delete logic in the intersection).

Algorithm 2 remove

Input: L a linked-list representing a set, e an element (node)
Output: none - e is removed to the set

```
?
```

Algorithm 3 size

Input: L a linked-list representing a set
Output: the number of elements in the set (0 if empty)

```
?
```

Algorithm 4 is_empty

Input: L a linked-list representing a set
Output: $true$ if the set is empty
?

Algorithm 5 union

Input: L_1 and L_2 two Linked-lists representing sets
Output: L_1 gets the union of the two sets (elements contained in any set without duplicates)
?

Algorithm 6 intersection

Input: L_1 and L_2 two Linked-lists representing sets
Output: L_1 gets the intersection of the two sets (elements contained in both sets)
?

Algorithm 7 difference

Input: L_1 and L_2 two Linked-lists representing sets
Output: L_1 gets the difference between A and B (elements that are in A but not in B)
?

Solutions

Algorithm 8 remove

Input: L a linked-list representing a set, e an element (node)
Output: none - e is removed from the set

```
found ← false
if L.element() = e.element() then
    L ← L.next()
else
    p ← L
    while p.next() is not null and found = false do
        if p.next().element() = e.element() then
            p.next() ← p.next().next()
            found ← true
        end if
        p ← p.next()
    end while
end if
```

Algorithm 9 size

Input: L a linked-list representing a set
Output: the number of elements in the set (0 if empty)

```
p ← L
n ← 0
while p is not null do
    n ← n + 1
    p ← p.next()
end while
return n
```

Algorithm 10 is_empty

Input: L a linked-list representing a set
Output: $true$ if the set is empty

```
return L = null
```

Algorithm 11 union

Input: L_1 and L_2 two Linked-lists representing sets
Output: L_1 gets the union of the two sets (elements contained in any set without duplicates)

```
p2 ← L2
while p2 is not null do
    found ← false
    p1 ← L1
    while p1 is not null and found = false do
        if p2.element() = p1.element() then
            found ← true
        else
            p1 ← p1.next()
        end if
    end while
    if found = false then
        e ← Node(p2.element())
        e.next() ← L1.head
        L1 ← e
    end if
    p2 ← p2.next()
end while
```

Algorithm 12 intersection

Input: L_1 and L_2 two linked-lists representing sets
Output: L_1 gets the intersection of the two sets (elements contained in both sets)

```
{This is the first part of the algorithm, which finds the first node that does not need to be removed (first element of the intersection)}
found ← false
while L1 is not null and found = false do
    p2 ← L2
    while p2 is not null and found = false do
        if L1.element() = p2.element() then
            found ← true
        else
            p2 ← p2.next()
        end if
    end while
    if found = false then
        L1 ← L1.next()
    end if
end while
{This is the second part of the algorithm: every time a node needs to be removed it is removed. We needed to make sure the first node of the list (if any) is not one to be removed though ;)}
if L1 is not null then
    p1 ← L1
    while p1.next() is not null do
        found ← false
        p2 ← L2
        while p2 is not null and found = false do
            if p1.next().element() = p2.element() then
                found ← true
            else
                p2 ← p2.next()
            end if
        end while
        if found = false then
            p1.next() ← p1.next().next()
        else
            if p1.next() is not null then
                p1 ← p1.next()
            end if
        end if
    end while
end if
```

Algorithm 13 intersection-lucas

Input: L_1 and L_2 two linked-lists representing sets
Output: L_1 gets the intersection of the two sets (elements contained in both sets)

{This is the core of the solution proposed: you create an "empty" new node at the start, pointing towards the head of the list, and you start the "checking" phase (the second part of the previous version of `intersection`) from this one.}
$temp \leftarrow newNode()$
$temp.next() \leftarrow L_1$
{Every time a node needs to be removed it is removed. Note that p_1 gets the address of the temporary node}
$p_1 \leftarrow temp$
while $p_1.next()$ is not null **do**
  $found \leftarrow false$
  $p_2 \leftarrow L_2$
  while p_2 is not null and $found = false$ **do**
    if $p_1.next().element() = p_2.element()$ **then**
      $found \leftarrow true$
    else
      $p_2 \leftarrow p_2.next()$
    end if
  end while
  if $found = false$ **then**
    $p_1.next() \leftarrow p_1.next().next()$
  else
    if $p_1.next()$ is not null **then**
      $p_1 \leftarrow p_1.next()$
    end if
  end if
end while
{Now we have to make sure L_1 is pointing to the "first element" of the list (the previous first element might have been deleted)}
$L_1 \leftarrow temp.next()$

Here is what happens with Lucas' intersection algorithm:

1. Figure 4.1 shows the situation at the start (L_1 has 3 elements and L_2 has one)

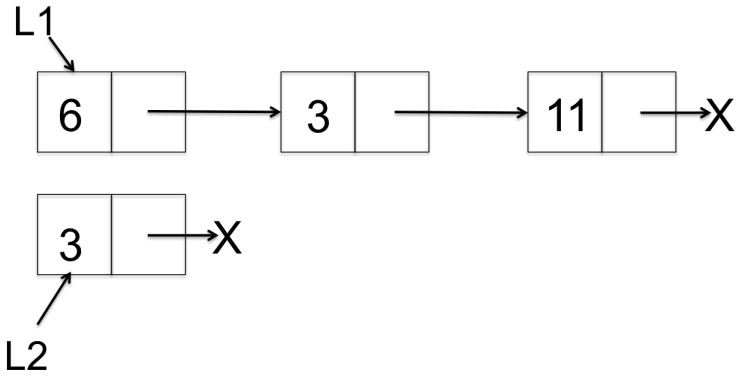


Figure 4.1: 1

2. Figure 4.2 shows the creation of the temp element

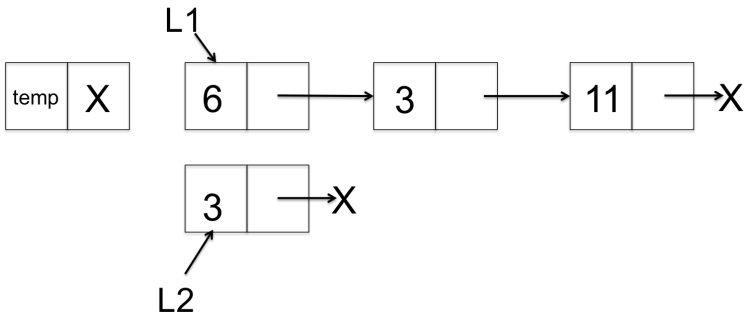


Figure 4.2: 2

3. Figure 4.3 shows temp being linked to the rest of the list

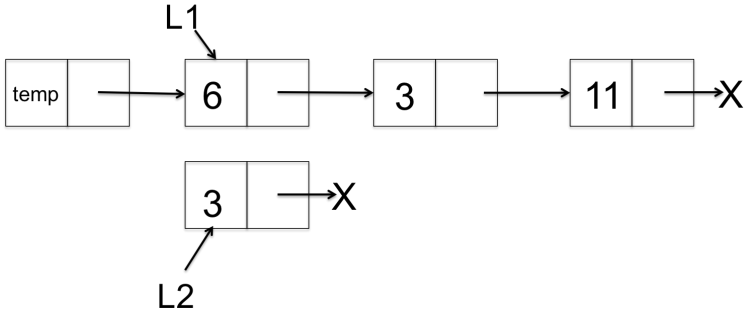


Figure 4.3: 3

4. Figure 4.4 shows p_1 being an address to the temp element (the "first" of the list). $p_1.next().element()$ (the value of the successor of p_1) is 6 which is not equal to one value of one node of L_2.

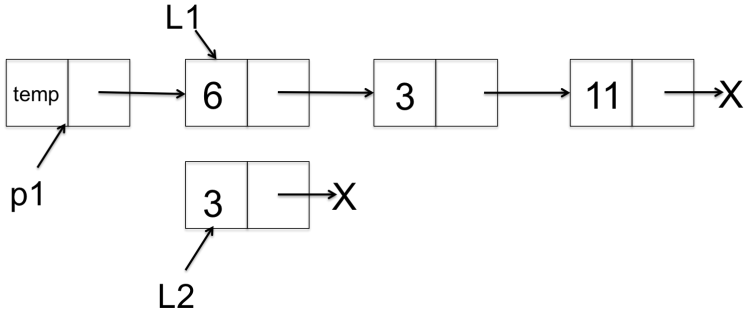


Figure 4.4: 4

5. Figure 4.5 shows how we "remove" the node in L_1 with the value 6 (the previous first node of the list, and the node that is the current p_1's successor). Note that p_1 does not change at this stage. Now we can also compare $p_1.next().element()$ (3) and L_2's values. There is a match so we keep it.

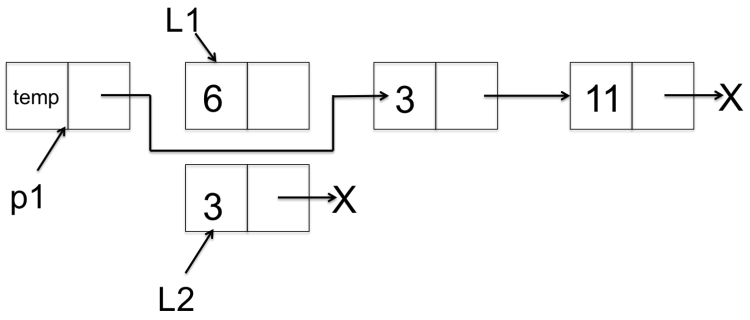


Figure 4.5: 5

6. Figure 4.6 shows that p_1 then moves to $p_1.next()$. Now we compare $p_1.next().element()$ (11) and L_2's values. There is no match so we'll have to remove it

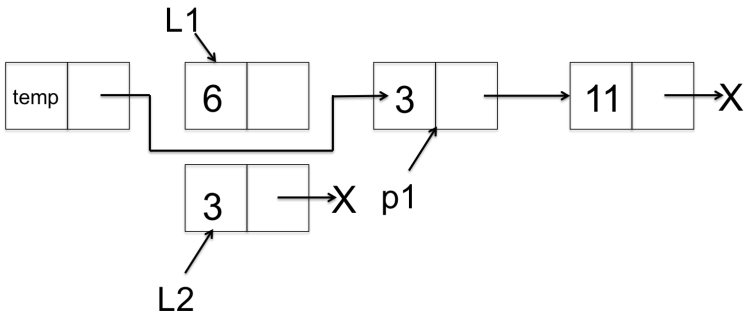


Figure 4.6: 6

7. Figure 4.7 shows how $p_1.next()$ (which is also node 3's next()) gets $p_1.next().next()$ Now the condition "$p_1.next()$ is not null" is false so we exit the main while loop

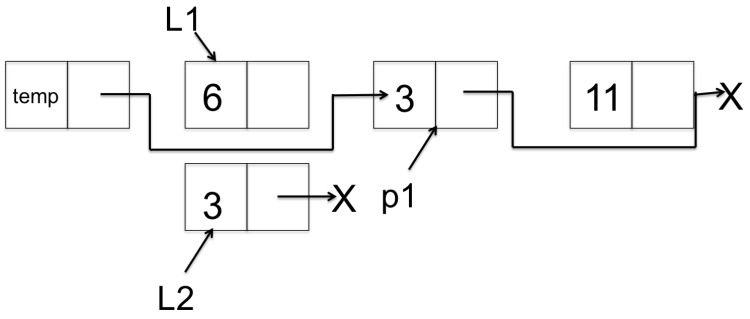


Figure 4.7: 7

8. Finally Figure 4.8 shows us the last thing to do: move L_1 to $temp.next()$, the first element after $temp$.

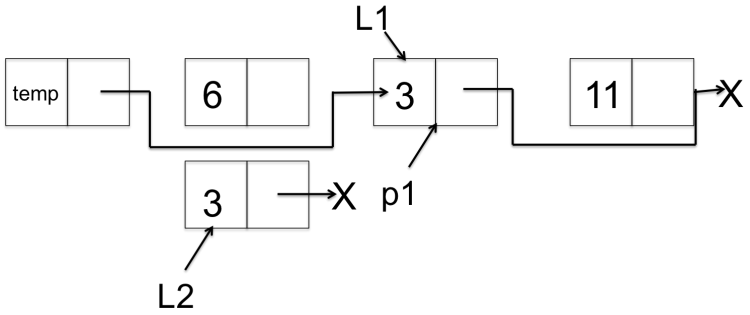


Figure 4.8: 8

9. Figure 4.9 shows you what's left of L_1: one single node (3) which is the intersection between L_1 and L_2

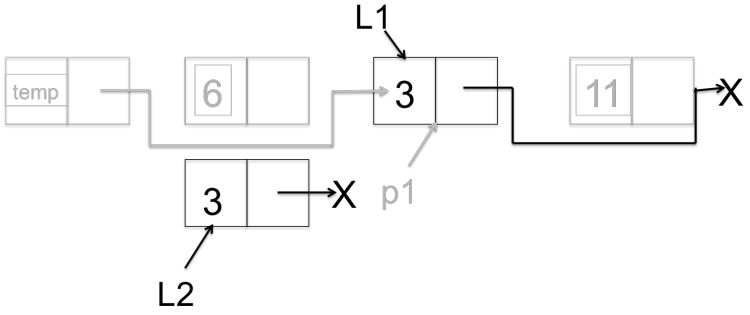


Figure 4.9: 9

Algorithm 14 difference

Input: L_1 and L_2 two linked-lists representing sets
Output: L_1 gets the difference between the two sets (elements contained only in L_1)

```
{Note that this algorithm is exactly the opposite (check the found = true/false conditions) as the previous one – we want here the "opposite" of the intersection.}
{This is the first part of the algorithm, which finds the first node that does not need to be removed (first element of the intersection)}
found ← true
while L1 is not null and found = true do
  found ← false
  p2 ← L2
  while p2 is not null and found = false do
    if L1.element() = p2.element() then
      found ← true
    else
      p2 ← p2.next()
    end if
  end while
  if found = true then
    L1 ← L1.next()
  end if
end while
{This is the second part of the algorithm: every time a node needs to be removed it is removed. We needed to make sure the first node of the list (if any) is not one to be removed though ;)}
if L1 is not null then
  p1 ← L1
  while p1.next() is not null do
    found ← false
    p2 ← L2
    while p2 is not null and found = false do
      if p1.next().element() = p2.element() then
        found ← true
      else
        p2 ← p2.next()
      end if
    end while
    if found = true then
      p1.next() ← p1.next().next()
    else
      if p1.next() is not null then
        p1 ← p1.next()
      end if
    end if
  end while
end if
```