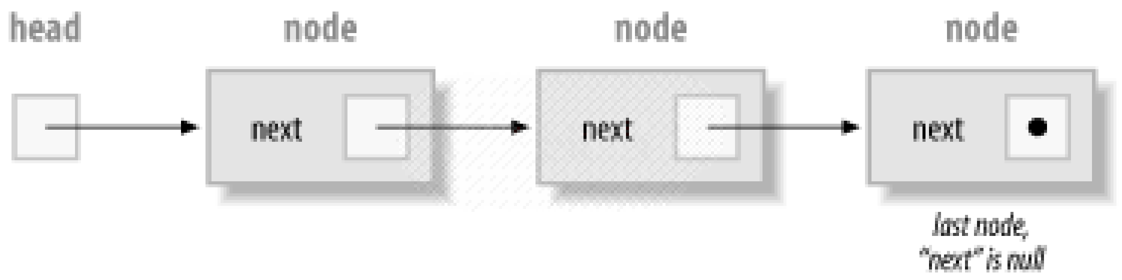
3. Linked Lists

3. Introduction to Linked Lists

- An array is a very useful data structure provided in programming languages.
- It has at least two limitations:
  - (1) changing the size of the array requires creating a new array and then copying all data from the array with the old size to the array with the new size
  - (2) The data in the array are next to each other sequentially in memory, which means that inserting an item inside the array requires shifting some other data in this array.
- These limitations can be overcome by using *linked structures.*
- A linked structure is a collection of nodes storing data and links to other nodes.
- In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the reference(s) to other node(s) in the structure.
- Although linked structures can be implemented in a variety of ways, the most flexible implementation is by using a separate object for each node.

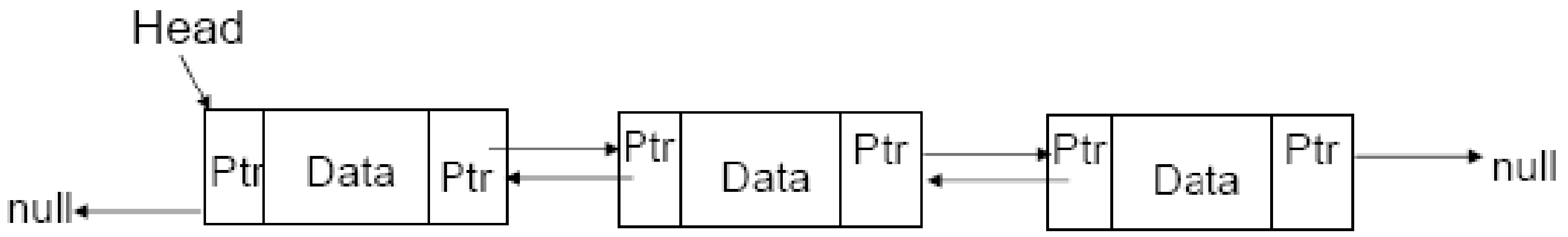
3.1. Singly linked lists



- If a node contains a data field that is a reference to another node, then many nodes can be used together using only one variable to access the entire sequence of nodes.
- Such a sequence of nodes is the most frequently used implementation of a *linked list,* which is a data structure composed of nodes, each node holding some information and a reference to another node in the list.
- If a node has a link only to its successor in this sequence, the list is called a *singly linked list.*
- Note that only one variable p is used to access any node in the list.
- The last node on the list can be recognized by the null reference field.

3.2. Doubly Linked Lists

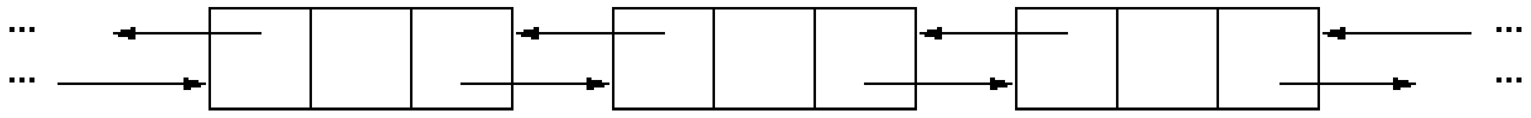
- Each node has a reference or pointer back to the previous nodes



3.2. Doubly Linked Lists

- If we wish to traverse a list both forwards and backwards efficiently, or if we wish, given a list element, to determine the preceding and following elements quickly, then the *doubly-linked list* comes in handy. A list element contains the data plus pointers to the next and previous list items as shown in the picture below.

A Doubly-Linked List



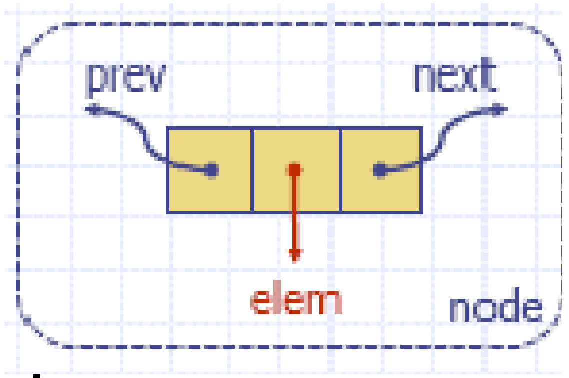
3.2. Doubly Linked List contd…

Example:
The full example can be found in the directory:
/home/331/tamj/examples/lists/doublyLinked

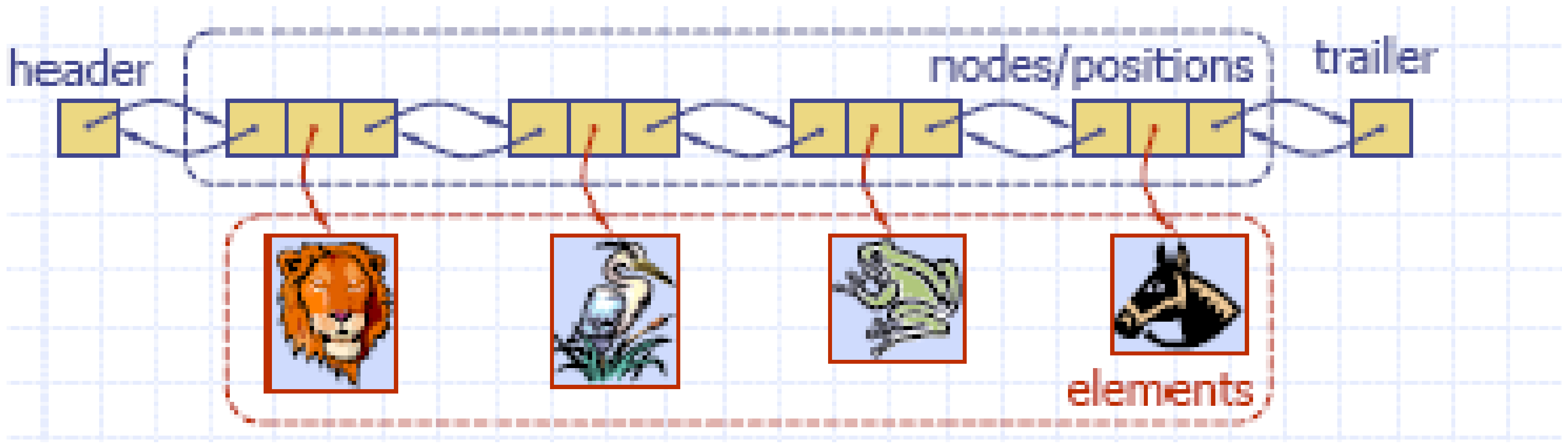
```
class ListManager
{
        private Node head;
        private int length;
        private int currentDataValue = 10;
        private static final int MAX_DATA = 100;
        : : : :
}
```

3.2. More about Doubly Linked List

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



3.2. More about Doubly Linked List contd….



3.2.1. Adding a new node at the end of a doubly linked list

- To add a node to a list, the node has to be created, its fields properly initialized, and then the node needs to be incorporated into the list.
- The process of Inserting a node at the end of a doubly linked list is performed in six steps:

1. A new node is created, and then its three fields are initialize as exaples info,el and prev.
2. the info field to the number el being inserted
3. the next field to null
4. and the prev field to the value of tail so that this field points to the last node in the list. But now, the new node should become the last node; therefore,
5. tail is set to reference the new node. But the new node is not yet accessible from its predecessor; to rectify this,
6. the next field of the predecessor is set to reference the new node.

3.2.1. Adding a new node at the end of a doubly linked list

- A special case concerns the last step. It is assumed in this step that the newly created node has a predecessor, so it accesses its prev field.
- It should be obvious that for an empty linked list, the new node is the only node in the list and it has no predecessor.
- In this case, both head and tail refer to this node, and the sixth step is now setting head to refer to this node.
- Note that step four—setting the prev field to the value of tail—is executed properly because for an initially empty list, tail is null. Thus, null becomes the value of the prev field of the new node.

3.2.1. Doubly Linked List: Adding To The End

```
public void addToEnd ()
{
        Node anotherNode = new Node (currentDataValue);
        Node temp;

                if (isEmpty() == true)
                   head = anotherNode;
                else {
                  temp = head;
            while (temp.next != null){
                temp = temp.next;
            }
            temp.next = anotherNode;
            anotherNode.previous = temp;
          }
          currentDataValue += 10;
          length++;
       }
```

3.2.2. Doubly Linked List: Adding Anywhere(1)

```
public void addToPosition (int position)
{
    Node anotherNode = new Node (currentDataValue);
    Node temp;
    Node prior;
    Node after;
    int index;
    if ((position < 1) || (position > (length+1)))
    {
        System.out.println("Position must be a value between 1-" +
        (length+1));
}
```

3.2.2. Doubly Linked List: Adding Anywhere(2)

```
else
{
    // List is empty
    if (head == null)
    {
        if (position == 1)
        {
            currentDataValue += 10;
            length++;
            head = anotherNode;
        }
    else
    System.out.println("List empty, unable to add node to " +"position " +
    position);
    }
```

3.2.2. Doubly Linked List: Adding Anywhere(3)

```
// List is not empty, inserting into first position.
else if (position == 1)
{
head.previous = anotherNode;
anotherNode.next = head;
head = anotherNode;
currentDataValue += 10;
length++;
}
```

3.2.2. Doubly Linked List: Adding Anywhere (4)

```
// List is not empty inserting into a position other than the first
else
{
    prior = head;
    index = 1;
    // Traverse list until current is referring to the node in front
    // of the position that we wish to insert the new node into.
    while (index < (position-1))
    {
        prior = prior.next;
        index++;
    }
    after = prior.next;
```

3.2.2. Doubly Linked List: Adding Anywhere (5)

```
    // Set the references to the node before the node to be
    // inserted.
    prior.next = anotherNode;
    anotherNode.previous = prior;

    // Set the references to the node after the node to be
    // inserted.
    if (after != null)
        after.previous = anotherNode;
    anotherNode.next = after;
    currentDataValue += 10;
    length++;
    }
  }
}
```

3.2.3. Deleting the last node from the doubly linked list

- Deleting the last node from the doubly linked list is straightforward because there is direct access from the last node to its predecessor, and no loop is needed to remove the last node.
- When deleting a node from the list, the temporary variable *el* is set to the value in the last node, then tail is set to its predecessor, and the last node is cut off from the list by setting the next field of the next to last node to null.
- In this way, the next to last node becomes the last node, and the formerly last node is abandoned.
- Although this node accesses the list, the node is inaccessible from the list; hence, it will be claimed by the *garbage* collector.
- The last step is returning the value stored in the removed node.

3.2.3. Deleting the last node from the doubly linked list

- An attempt to delete a node from an empty list may result in a program crash.
- Therefore, the user has to check whether the list is not empty before attempting to delete the last node.
- As with the singly linked list's deleteFromHead(), the caller should have an if statement

```
if   (!list.isEmpty())
     n = list.deleteFromTail();
else do not remove;
```

3.2.3. Deleting the last node from the doubly linked list

- The second special case is the deletion of the only node from a single-node linked list. In this case, both head and tail are set to null.
- Because of the immediate accessibility of the last node, both addToTail () and deleteFromTail () execute in constant time O(l).
- Methods for operating at the beginning of the doubly linked list are easily obtained from the two methods discussed by changing head to tail and vice versa, changing next to prev and vice versa, and exchanging the order of parameters when executing new.

3.2.3. Doubly Linked List: Deleting A Node(1)

```
public void delete (int key)
{
    int indexToDelete;
    int indexTemp;
    Node previous;
    Node toBeDeleted;
    Node after;
    indexToDelete = search(key);
    // No match, nothing to delete.
    if (indexToDelete == -1)
    {
                        System.out.println("Cannot delete element with a
                data value of "
                        + key + " because it was not found.");
    }
```

3.2.3. Doubly Linked List: Deleting A Node(2)

```
else
{
        // Deleting first element.
        if (indexToDelete == 1)
        {
                head = head.next;
                length--;
        }
        else
        {
                                previous = null;
                                toBeDeleted = head;
                                indexTemp = 1;
                                while (indexTemp < indexToDelete)
                                {
                previous = toBeDeleted;
                toBeDeleted = toBeDeleted.next;
                indexTemp++;
                }
                previous.next = toBeDeleted.next;
                after = toBeDeleted.next;
                after.previous = previous;
                length--;
                : : :
```

3.2.4. Pros Of Doubly Linked Lists

- Pros
  - Traversing the list in reverse order is now possible.
  - One can traverse a list without a trailing reference (or by scanning ahead)
  - It is more efficient for lists that require frequent additions and deletions near the front and back

3.2.5. Cons Of Doubly Linked Lists

- Cons
  - An extra reference is needed
  - Additions and deletions are more complex (especially near the front and end of the list)

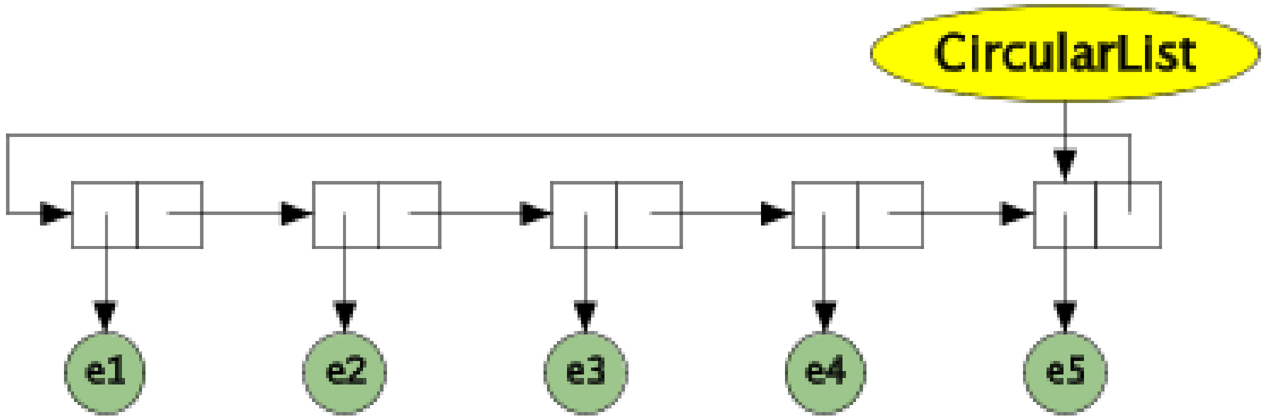
3.2.6. An implementation of a doubly linked list

```
/*************************IntDLLNode.java******************************/
public class IntDLLNode {
   public int info;
   public IntDLLNode next, prev;
   public IntDLLNode (int el) {
            this (el,null,null);
   }
   public IntDLLNode (int el, IntDLLNode n, IntDLLNode p) {
            info = el; next =n; prev=p;
   }
}
```

3.2.6. An implementation of a doubly linked list(2)

```
/**************************IntDLList.java*********************************/
public class IntDLList {
   private IntDLLNode head, tail;
   public IntDLList ( ) {
            head = tail = null; }
   public boolean isEmpty( ) {
            return head == null; }
   public void addToTail (int el) {
            if (!isEmpty ( )) {
                tail = new IntDLLNode (el, null, tail);
                tail.prev.next = tail; }
            else head = tail = new IntDLLNode(el);
   }
   public int removeFromTail ( ) {
            int el = tail.info;
            if (head == tail)        // if only one node in the list;
                 head = tail =null;
            else {                   // if more than one node in the list;
                 tail = tail.prev;
                 tail.next = null; }
            return el;
   }
   ………………………… }
```

3.3. Circular Lists



A circular list. The large yellow object represents the circular list as such. The circular green nodes represent the elements of the list. The rectangular nodes are instances of a class similar to LinkedListNode, which connect the constituents of the list together.

3.3. Circular Lists

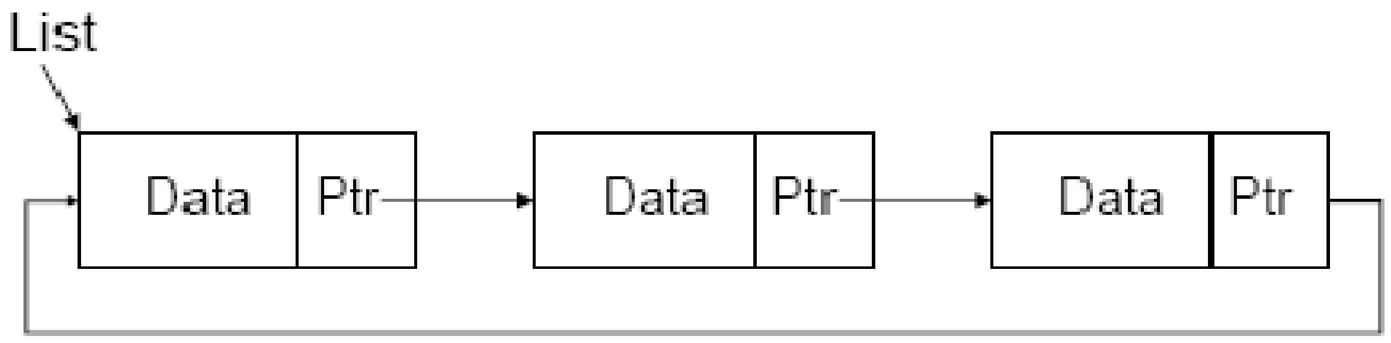
- A *circular list* is needed in which nodes form a ring: The list is finite and each node has a successor.
- An example of such a situation is when several processes are using the same resource for the same amount of time, and we have to assure that each process has a fair share of the resource.
- Therefore, all processes—let their numbers be 6, 5, 8, and 10, are put on a circular list accessible through current.
- After one node in the list is accessed and the process number is retrieved from the node to activate this process, current moves to the next node so that the next process can be activated the next time.

3.3. Circular Lists

- In an implementation of a circular singly linked list, we can use only one permanent reference, tail, to the list even though operations on the list require access to the tail and its successor, the head.
- To that end, a linear singly linked list uses two permanent references, **head** and **tail**.

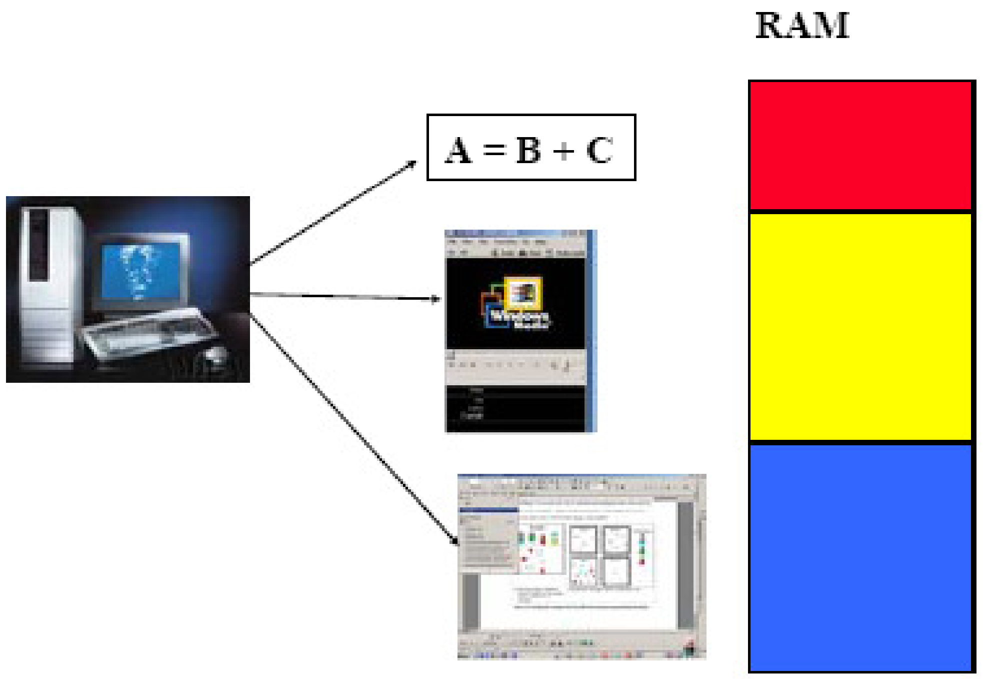
3.3. Circular Linked Lists

- An extra link from the end of the list to the front forms the list into a ring



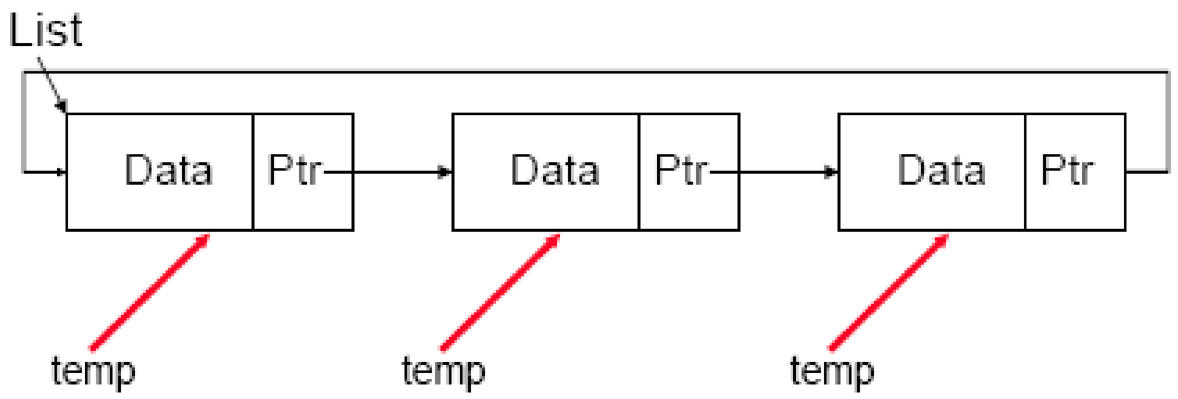
3.3.1. Uses Of A Circular List

- e.g., Memory management by operating systems



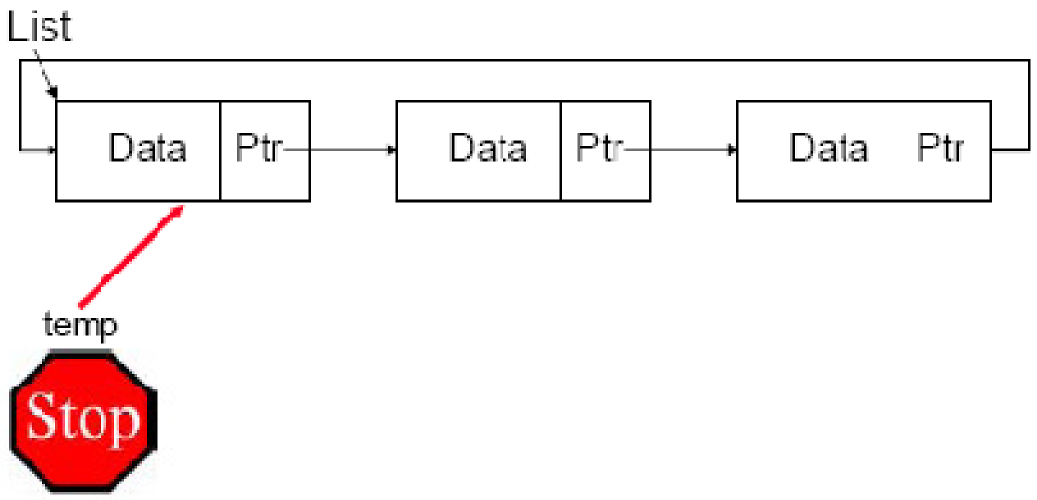
3.3.2. Searches With A Circular Linked Lists

- Cannot use a null reference as the signal that the end of the list has been reached.
- Must use the list reference as a point reference (stopping point) instead



3.3.3. Traversing A Circular Linked List

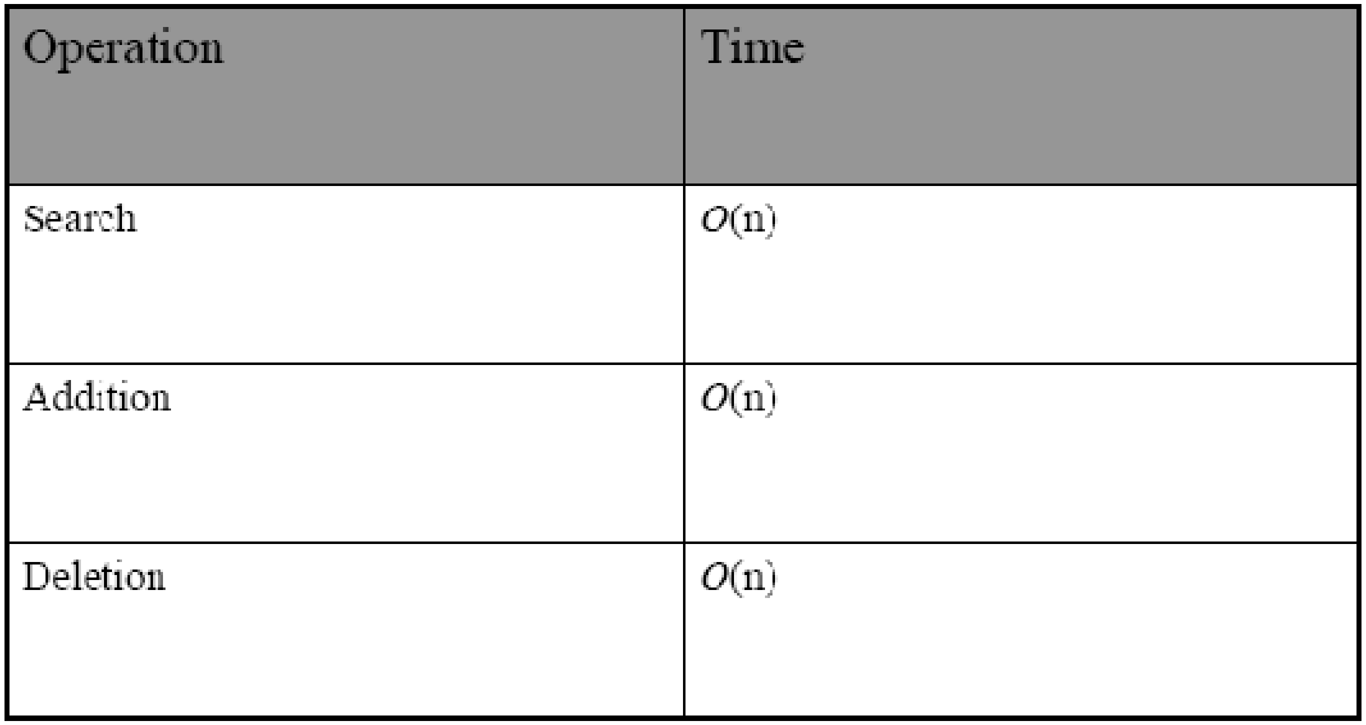
- Cannot use a null reference as the signal that the end of the list has been reached.
- Must use the list reference as a point reference (stopping point) instead



3.3.4. An Example Of Traversing A Circular Linked List

```
public void display ()
{
        Node temp = list;
        System.out.println("Displaying list");
        if (isEmpty() == true)
        {
          System.out.println("Nothing to display, list is empty.");
        }
        do
        {
          System.out.println(temp.data);
          temp = temp.next;
        } while (temp != list);
          System.out.println();
}
```

3.3.5. Worse Case Times For Circular Linked Lists



Operation	Time
Search	O(n)
Addition	O(n)
Deletion	O(n)

3.4. Skip Lists

- Linked lists have some serious drawbacks:
  - They require sequential scanning to locate a searched-for element.
  - The search starts from the beginning of the list and stops when either a searched-for element is found or the end of the list is reached without finding this element.
  - Ordering elements on the list can speed up searching, but a sequential search is still required. Therefore, we may think about lists that allow for skipping certain nodes to avoid sequential processing.
  - A *skip list* is an interesting variant of the ordered linked list that makes such a nonsequential search possible (Pugh 1990).

3.4. Skip Lists contd…

- In a skip list of n nodes, for each k and i such that $1 <= k <=$ [log n] and $1 <= i <= [n/2^{k-1}] - 1$, the node in position $2^{k-1} \bullet i$ points to the node in position $2^{k-1} \bullet (i + 1)$.
- This means that every second node points to the node two positions ahead, every fourth node points to the node four positions ahead, and so on.
- This is accomplished by having different numbers of reference fields in nodes on the list:
  - Half of the nodes have just one reference field
  - one-fourth of the nodes have two reference fields
  - one-eighth of the nodes have three reference fields and etc.
- The number of reference fields indicates the *level* of each node, and the number of levels is $maxLevel = [\lg n] + 1$.

3.5. Self-organizing lists

- The introduction of skip lists was motivated by the need to speed up the searching process.
- Although singly and doubly linked lists require sequential search to locate an element or to see that it is not in the list, we can improve the efficiency of the search by dynamically organizing the list in a certain manner.
- This organization depends on the configuration of data; thus, the stream of data requires reorganizing the nodes already on the list.
- There are many different ways to organize the lists, and this section describes four of them:
  – *Move-to-front method.* After the desired element is located, put it at the beginning of the list.
  – *Transpose method.* After the desired element is located, swap it with its predecessor unless it is at the head of the list.
  – *Count method.* Order the list by the number of times elements are being accessed.
  – *Ordering method.* Order the list using certain criteria natural for the information under scrutiny.

3.5. Self-organizing lists contd…

- In the first three methods, new information is stored in a node added to the end of the list;
- In the fourth method, new information is stored in a node inserted somewhere in the list to maintain the order of the list.

3.5. Self-organizing lists contd…

- With the first three methods, we try to locate the elements most likely to be looked for near the beginning of the list, most explicitly with the move-to-front method and most cautiously with the transpose method.
- The ordering method already uses some properties inherent to the information stored in the list.
  For example, if we are storing nodes pertaining to people, then the list can be organized alphabetically by the name of the person or the city or in ascending or descending order using, say, birthday or salary.
- This is particularly advantageous when searching for information that is not in the list, because the search can terminate without scanning the entire list.
- Searching all the nodes of the list, however, is necessary in such cases using the other three methods.
- The count method can be subsumed in the category of the ordering methods if frequency is part of the information.
- In many cases, however, the count itself is an additional piece of information required solely to maintain the list; hence, it may not be considered "natural" to the information at hand.

3.5. Self-organizing lists contd…

- Analyses of the efficiency of these methods customarily compare their efficiency to that of *optimal static ordering.*
- With this ordering, all the data are already ordered by the frequency of their occurrence in the body of data so that the list is used only for searching, not for inserting new items.
- Therefore, this approach requires two passes through the body of data,
  - one to build the list
  - another to use the list for search alone.