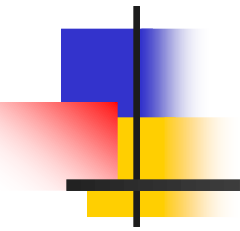
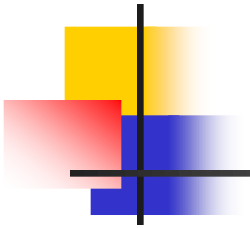


Chapter 5

LINKED LISTS



Objective

- To introduce:
  - Concept and linked lists statements
  - Types of linked lists
  - Representation of linked lists in memory
  - Linked lists operations

CONTENT

5.1 Introduction
5.2 Types of Linked Lists
5.3 Representation of Linked Lists in Memory
5.4 A Pointer-Based Implementation of Linked Lists
5.5 Linked Lists Application

5.1 Introduction

A. LINEAR LIST CONCEPTS

- The simplest linear list structure, the array, is found in virtually all programming languages. The sequentially of a linear list is diagrammed in Figure 5.1.

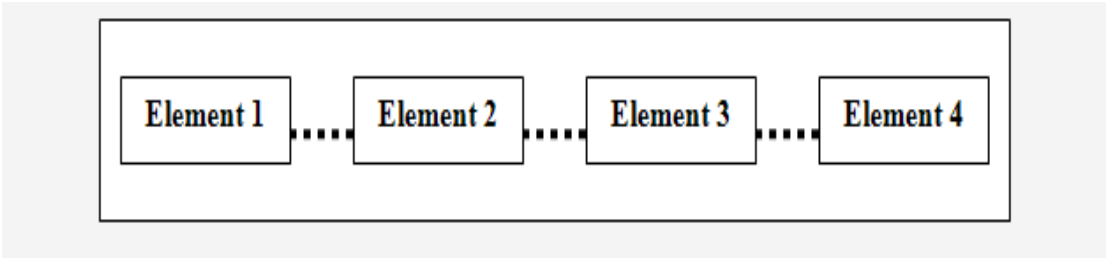
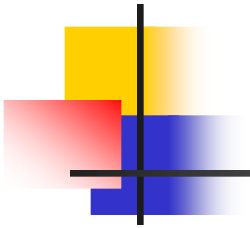


Figure 5.1 A Linear List



5.1 Introduction

- Linear lists can be divided into two categories: general and restricted.

- In a general list, data can be inserted and deleted anywhere and there are no restrictions on the operations that can be used to process the list. General structures can be further described by their data as either random or ordered lists. In a random list, there is no ordering of the data. In an ordered list, the data are arranged according to a key. A key is one or more fields within a structure that are used to identify the data or otherwise control their use. In the simple array, the data are also the keys. In an array of records structure, the key is a field, such as employee number, that identifies the record.

5.1 Introduction

- In a restricted list, data can only be added or deleted at the ends of the structure and processing is restricted to operations on the data at the ends of the list. There are two restricted list structures: the first in-first out (FIFO) list and the last in-first out (LIFO) list. The FIFO list is generally called a queue; the LIFO list is generally called a stack. Figure 5.2 shows types of lists.

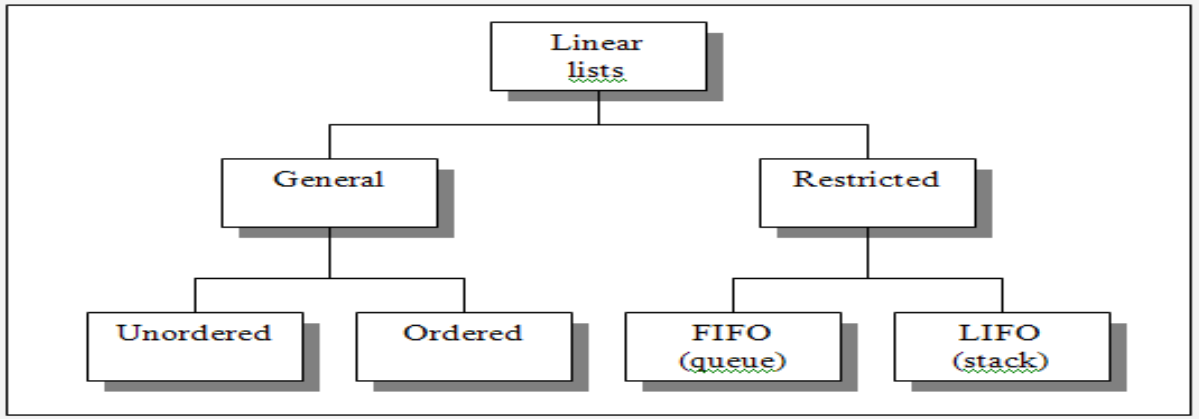
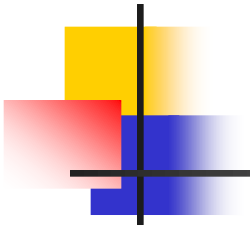


Figure 5.2 Types of Lists



5.1 Introduction

- Four operations are generally associated with linear lists: **insertion, deletion, retrieval** and **traversal**. Insertion, deletion and retrieval apply to all lists; traversal is not applicable to restricted list.

5.1 Introduction

B. LINKED LIST CONCEPTS

- It is an ordered collection of data in which each element contains the location of the next element. Each element is referred to as node.
- Each node contains two fields as shown in Figure 5.3:
  - ✓ data – holds useful information, the data to be processed.
  - ✓ link – contains a pointer that identifies the next element in the list. In addition, a pointer variable identifies the first element in the list. The name of the list is the same as the name of the pointer variable.

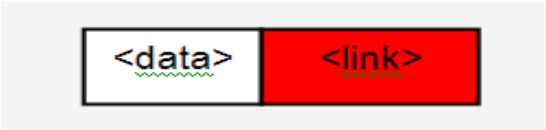


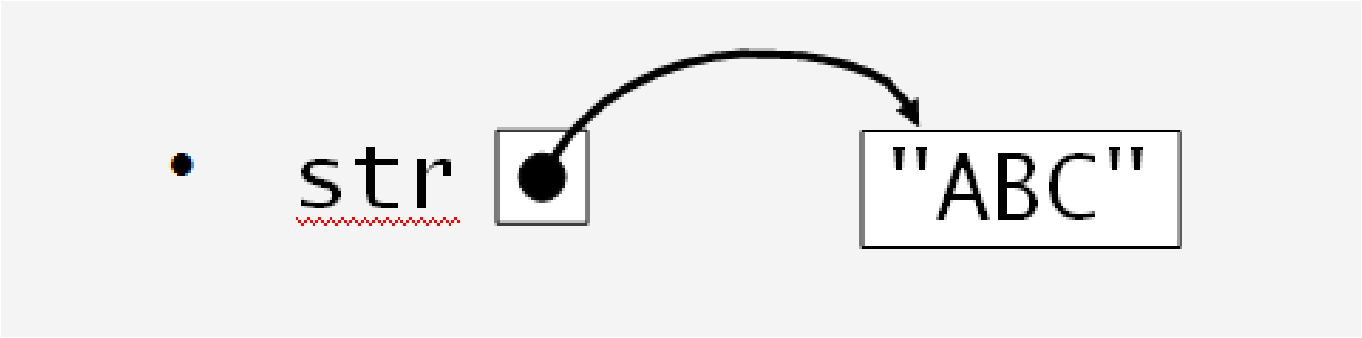
Figure 5.3 A node with a data field and a link.

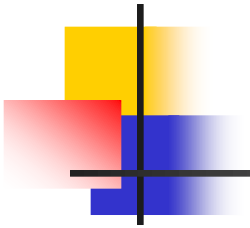
5.1 Introduction

- The simple linked list described here is commonly known as a singly linked list because it contains only one link to a single successor.
- Advantage – data are easily inserted and deleted without shifting elements of a linked list.
- Disadvantage – limited to sequential searches and retrieving elements.
- The graphic  represents a *pointer*, which is a memory address of an object.
- The object that it points to is the object whose memory address is stored.
- The graphic  o  represents the *null pointer*, whose memory address is 0. It points to nothing. (No object can be stored at 0.)

5.1 Introduction

- In Java, a variable that points to an object is called a *reference* variable.
- String str = "ABC";
- Here, **str** is a reference variable that refers to a **String** object that contains "ABC".





5.1 Introduction

- Data for each node in a linked list contains the same data type (i.e. simple data type or structural) as in Figure 5.4.
- In Figure 5.4, the first node contains a single field, number, and a link. The second is more typical. It contains three data fields, a name, id, and grade points (grdPts), and a link. The third is recommended. The fields are defined in their own structure, which is then put into the definition of a node structure. The one common element in all examples is the link field.
- The nodes in a linked list are called self-referential structures. In a self-referential structure, each instance of the structure contains a pointer to another instance of he same structural type. In Figure 5.4, the shaded boxes with arrows are the links that make the linked list a self-referential structure.

5.1 Introduction

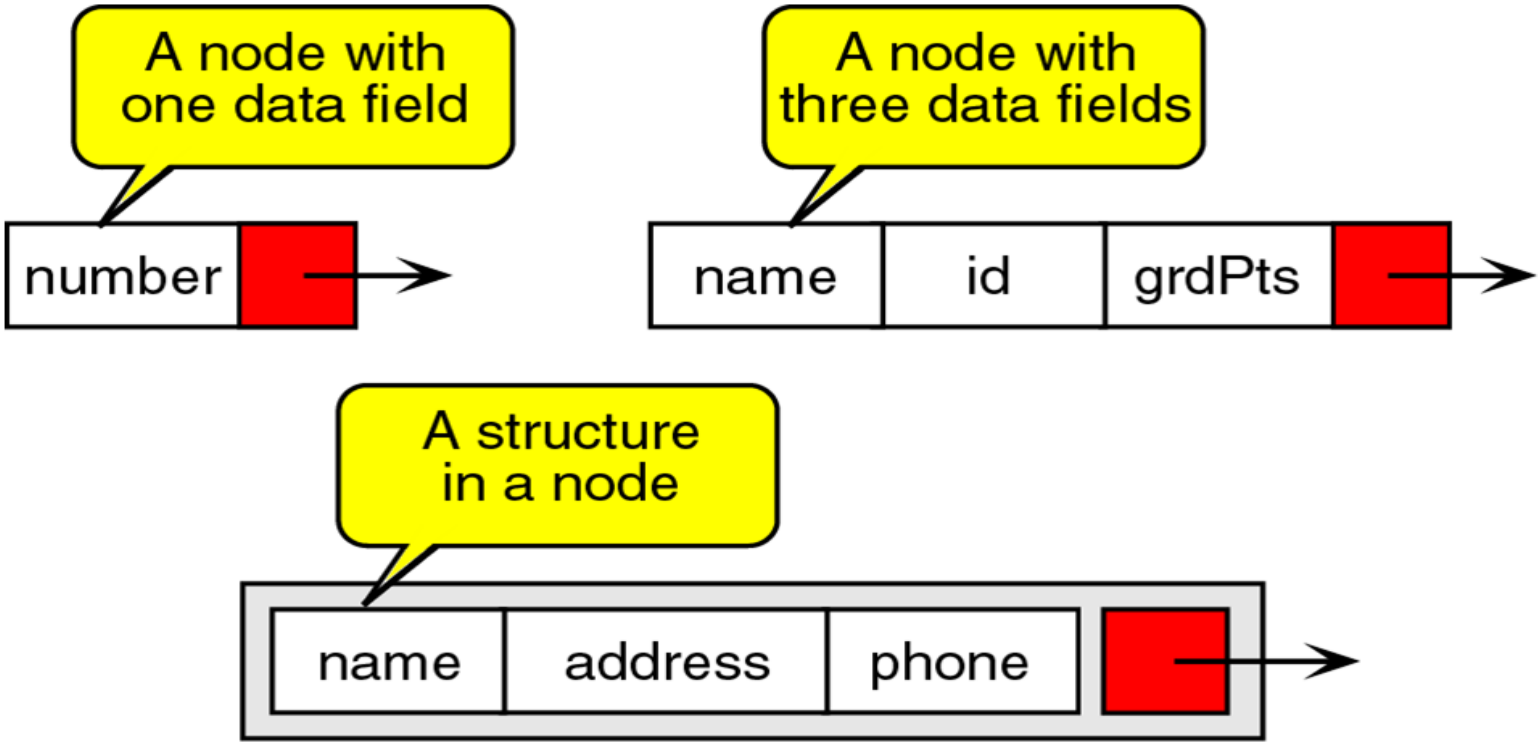
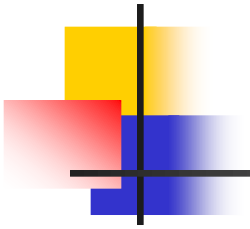
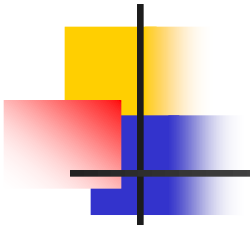


Figure 5.4: Nodes



5.1 Introduction

- Basic operations:
  - ✓ Construction – create an empty list.
  - ✓ isEmpty – check if the list is empty; TRUE if it is empty, while FALSE if it is otherwise.
  - ✓ Insert – insert node into the list.
  - ✓ Delete – remove node from the list.
  - ✓ Traverse – go through the list of a part of it, accessing and processing the elements.
  - ✓ Search – locate data in a list.



5.1 Introduction

- Java Declaration:

```
private class Node {
    Object data;
    Node link;

    public Node(Object o) {
      data = o;
    }
 }
```

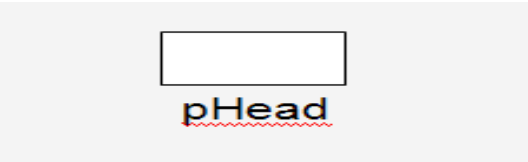
5.1 Introduction

Example:

Node pHead;

- This declaration will allocate a reference variable **pHead**.

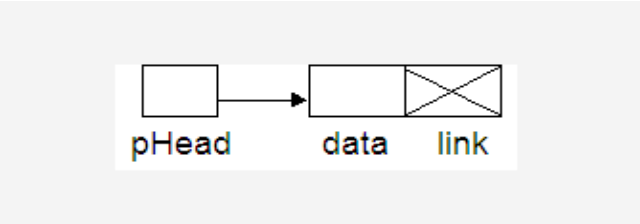
  In Java, each reference variable either locates an object or it is null.



- Statement

pHead = new Node();

dynamically will allocate a new memory cell of type Node. Link **pHead** will point to the new cell.



5.1 Introduction

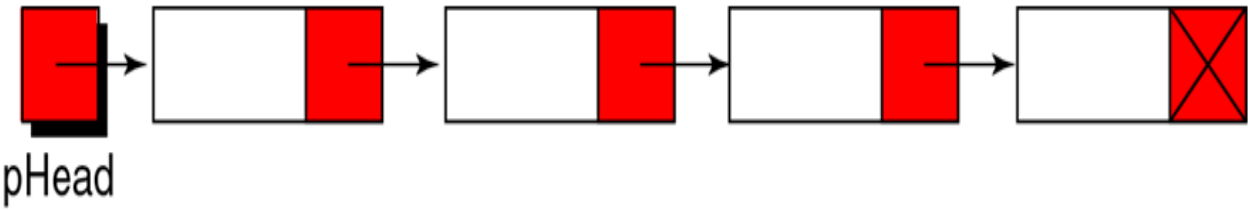


Figure 5.5: A linked list.

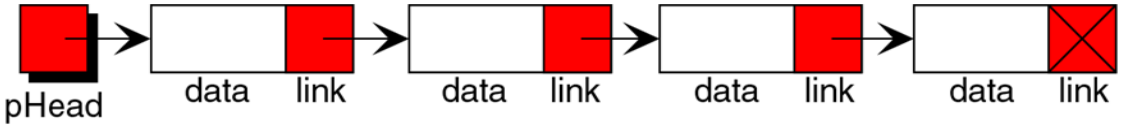
- Statements
  pHead.data
  pHead.link

- Statement
  pHead = null;

will refer to the link that points to an empty list. (i.e. Figure 5.6(b)). The link will always exist even though there is no node in the linked list. It is because the link has been declared.

5.1 Introduction

- A linked list is referred to using external link. (i.e. pHead, Figure 5.6(a)).
- External link (i.e. pHead, Figure 5.6(a)) store the location of the first node. It is not a node in the linked list and do not store any data.



(a) A linked list with a head pointer: pHead

(b) An empty linked list

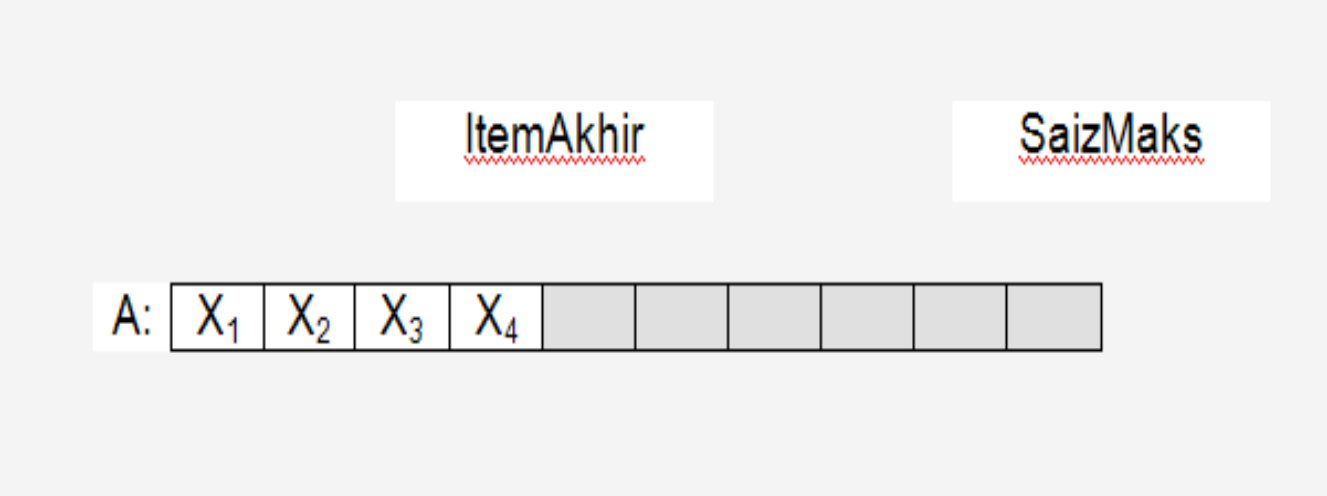
Figure 5.6 A Linked List

- A link that builds relation between the linked list nodes is called an internal link, i.e. the next member to the node - link (next member)).

5.2 Types of Linked Lists

a) Sequential List

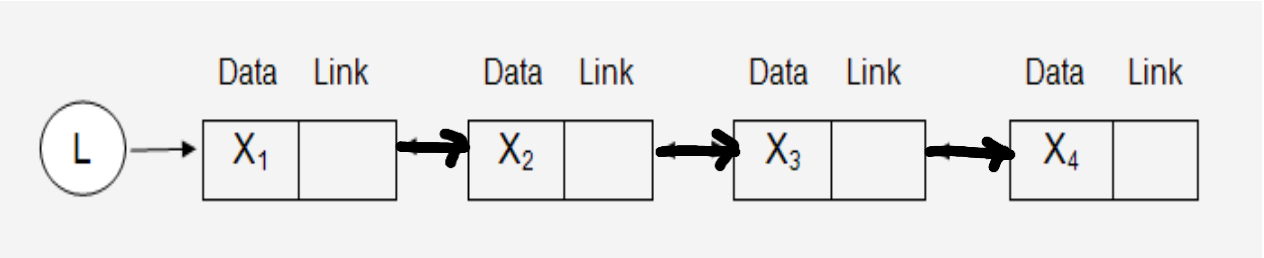
- ✓ Using an array.
- ✓ To determine the list size, index is kept for the last item. (i.e. ItemAkhir). This list contains features of an array.



5.2 Types of Linked Lists

b) Singly Linked List

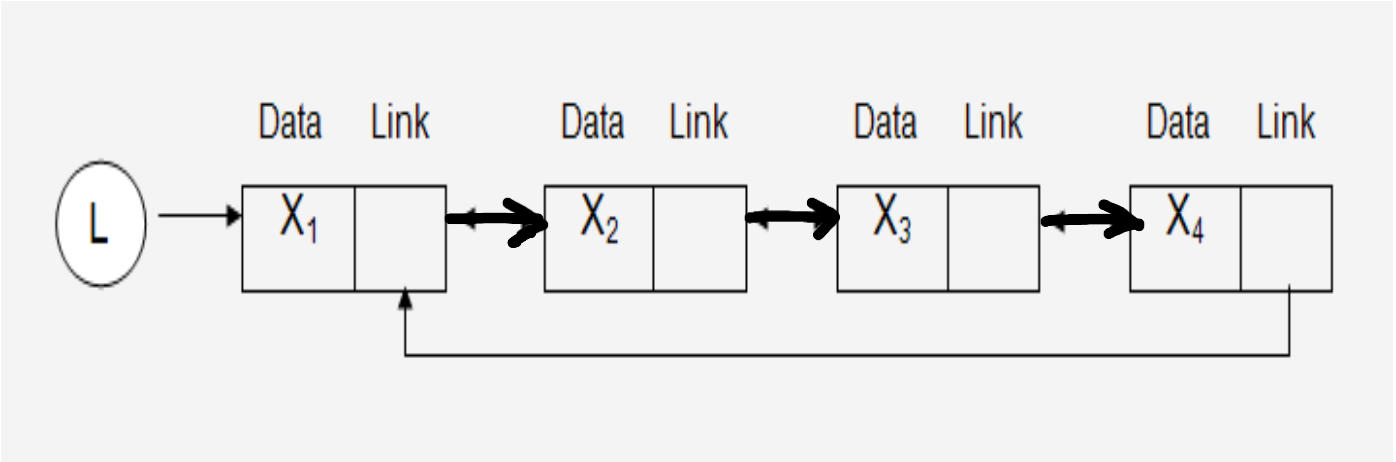
- ✓ Using a link.
- ✓ The operations are create list, insert node, delete node, search list, unordered list search, empty list, full list, list count, traverse list and destroy list.



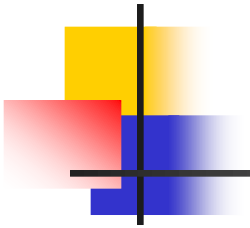
5.2 Types of Linked Lists

c) Circularly Linked List

- ✓ A linked list where the last node's link points to the first node of the list.



- ✓ Advantage: allow access to nodes in the middle of the list without starting at the beginning.



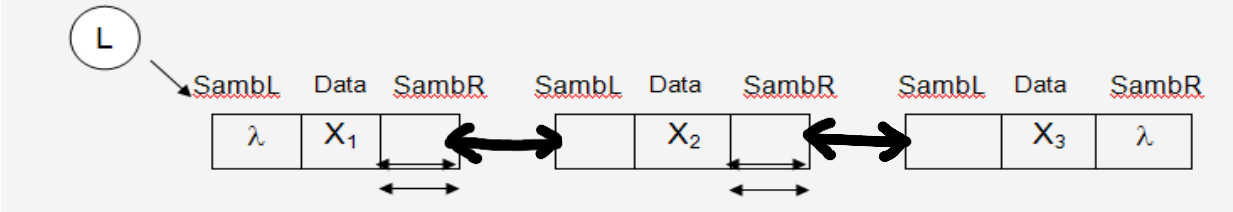
5.2 Types of Linked Lists

- ✓ Insertion and deletion into a circularly linked list follow the same logic patterns used in a singly linked list except that the last node points to the first node. Therefore, when inserting or deleting the last node, in addition to updating the rear pointer in the header, we must also point the link field to the first node.
- ✓ In a singly linked list, when we arrive at the end of the list the search is complete. In a circular list, however, we automatically continue the search from the beginning of the list.
- ✓ In the singly linked list, if we didn't find the data we were looking for, we stopped when we hit the end of the list or when the target was less than the current node's data. With a circular list, we save the starting node's address and stop when we have circled around to it.

5.2 Types of Linked Lists

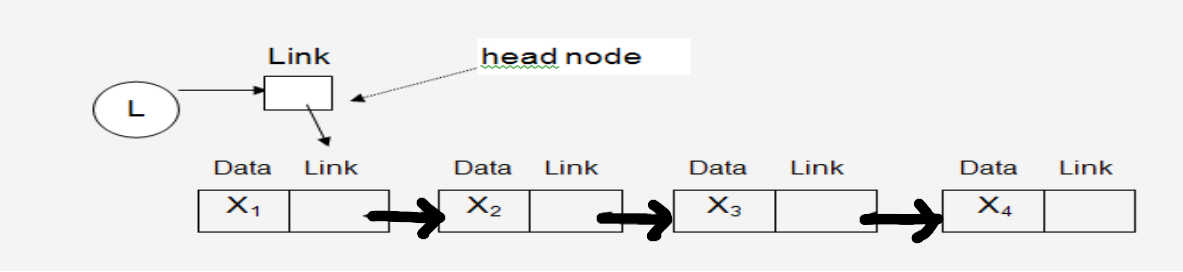
d) Doubly Linked List

- ✓ Each node has a pointer to both its successor and its predecessor.



e) Dummy Head Node

- ✓ It is a one-way linked list but contains a unique head node that points to the first node.



5.2 Types of Linked Lists

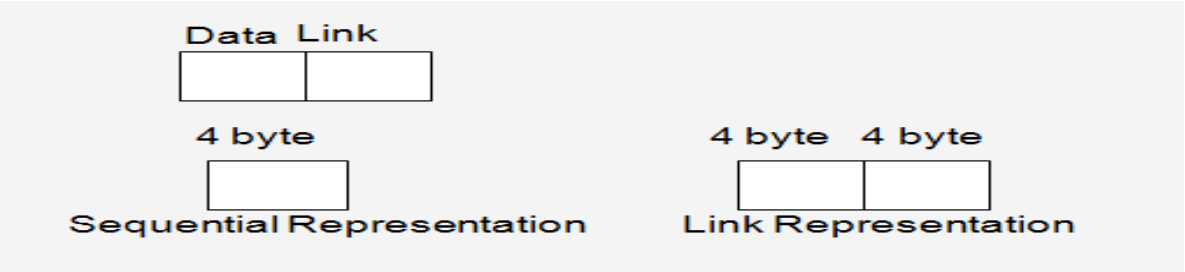
Differences between sequential representation and link:

- In terms of time, the difference is computed using complexity.

  Besides time factor, storage factor and cost factor need to be considered.

- Example:

  A link representation requires an extra space for link of each node.



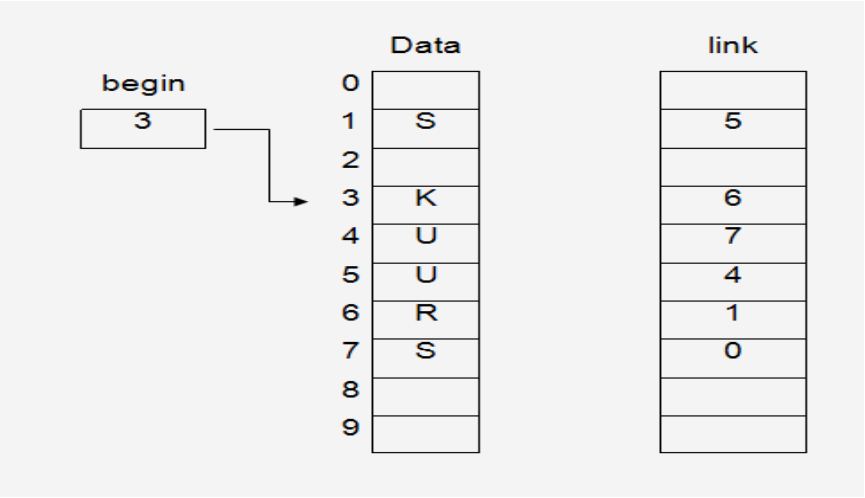
- In a sequential representation, an unused declared space will incur a waste.

5.3 Representation of Linked Lists In Memory

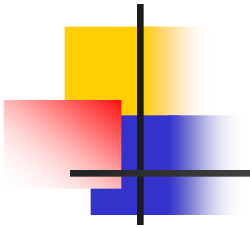
- Requires two linear arrays.
- Example:

  A list SEN requires 2 arrays, data[n] and link[n] with one variable to store the value of the initial location of the list.

  End value NULL: 0 or negative.

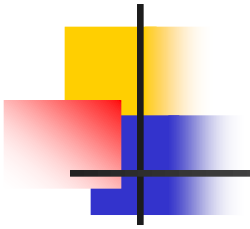


5.4 Common Behaviors / Methods of Linked Lists



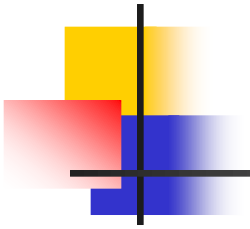
- Create List
  - ✓ Receives the head structure and initializes the metadata for the list.
- Destroy List
  - ✓ Destroy list deletes any nodes still in the list.
- Add new node
  - ✓ To add a new node to an empty list is to assign the list head pointer the address of the new node and make sure that its link field is a null pointer.
- Insert Node
  - ✓ Insert can be done at the beginning of the list, at the middle of the list or at the end of the list.

5.4 Common Behaviors / Methods of Linked Lists



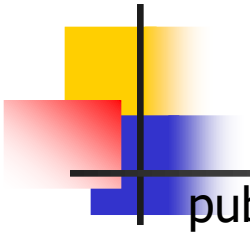
- Empty List
  - ✓ When the head pointer of the list is null, then the list is empty.
- Delete Node
  - ✓ Logically remove a node from the linked list by changing various link pointers and then physically deleting the node from dynamic memory.
  - ✓ Delete can be done at the first node, at the last node or at a specified position of the list.
- Traverse List
  - ✓ Algorithms that traverse a list start at the first node and examine each node in succession until the last node has been processed.

5.4 Common Behaviors / Methods of Linked Lists



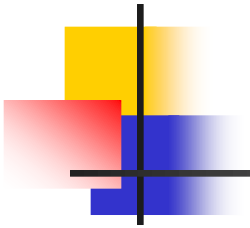
- Search List
  - ✓ Search algorithm is used by several algorithms to locate element in a list.
  - ✓ Sequential search is used because there is no physical relationship among the nodes.
  - ✓ The classics sequential search returns the location of an element when it is found.
- Length List
  - ✓ Counts the number of nodes in the list.

5.4 A Java Implementation of Linked Lists



```
public class MyLinkedList {
  Node first, last;
  public static int size;

  public MyLinkedList() {
  }

   public Object getFirst() {
    if (size == 0)
     return null;
    else
     return first.element;
  }
```

5.4 A Java Implementation of Linked Lists



```
public Object getLast() {
      if (size == 0)
       return null;
      else
     return last.element;
  }
```

5.4 A Java Implementation of Linked Lists

```
/** Add an element to the beginning of the list */
  public void addFirst(Object o) {
    Node newNode = new Node(o);
    newNode.next = first;
    first = newNode;
    size++;

    if (last == null)
      last = first;
  }
```

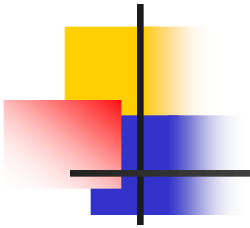
5.4 A Java Implementation of Linked Lists

```
/** Add an element to the end of the list */
  public void addLast(Object o) {
    if (last == null) {
      first = last = new Node(o);
    }
    else {
      last.next = new Node(o);
      last = last.next;
    }

    size++;
  }
```

5.5 A Java Implementation of Linked Lists

```
/** Adds a new element o at the specified index in this list
 * The index of the first element is 0 */
public void add(int index, Object o) {
  if (index == 0) addFirst(o);
  else if (index >= size) addLast(o);
  else {
    Node current = first;
    for (int i = 1; i < index; i++)
      current = current.next;
    Node temp = current.next;
    current.next = new Node(o);
    (current.next).next = temp;
    size++; }
}
```

5.5 A Java Implementation of Linked Lists



```
/** Remove the first node and
  *  return the object that is contained in the removed node. */
 public Object removeFirst() {
   if (size == 0) return null;
   else {
     Node temp = first;
     first = first.next;
     size--;
     if (first == null) last = null;
     return temp.element;
   }
 }
```

5.4 A Java Implementation of Linked Lists

```
/** Remove the last node and
  * return the object that is contained in the removed node. */
 public Object removeLast() {
   // Implementation left as an exercise
   return null;
 }
```

5.4 A Java Implementation of Linked Lists

```
/** Removes the element at the specified position in this list.
   *  Returns the element that was removed from the list. */
  public Object remove(int index) {
    if ((index < 0) || (index >= size)) return null;
    else if (index == 0) return removeFirst();
    else if (index == size - 1) return removeLast();
    else {
      Node previous = first;
```

5.4 A Java Implementation of Linked Lists

```
for (int i = 1; i < index; i++) {
        previous = previous.next;
      }

      Node current = previous.next;
      previous.next = current.next;
      size--;
      return current.element;
    }
  }
```

5.4 A Java Implementation of Linked Lists

```
/** Override toString() to return elements in the list */
public String toString() {
  StringBuffer result = new StringBuffer("[");
  Node current = first;
  for (int i = 0; i < size; i++) {
    result.append(current.element);
    current = current.next;
    if (current != null)
      result.append(", "); // Seperate two elements with a comma
    else
      result.append("]"); }// Insert the closing ] in the string
  return result.toString();
}
```

5.4 A Java Implementation of Linked Lists

```
/** Return true if this list contains the element o */
  public boolean contains(Object o) {
    // Implementation left as an exercise
    return true;
  }

   /** Return the element from this list at the specified index */
  public Object get(int index) {
    // Implementation left as an exercise
    return null;
  }
```

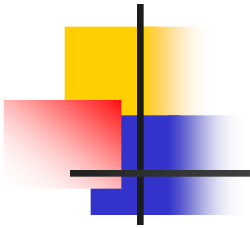
5.4 A Java Implementation of Linked Lists

```
/** Returns the index of the first matching element in this list.
 *  Returns -1 if no match. */
public int indexOf(Object o) {
  // Implementation left as an exercise
  return 0;
}

 /** Returns the index of the last matching element in this list
 *  Returns -1 if no match. */
public int lastIndexOf(Object o) {
  // Implementation left as an exercise
  return 0;
}
```

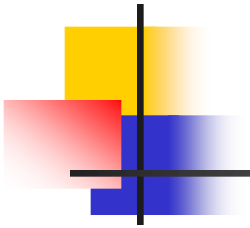
5.4 A Java Implementation of Linked Lists

```
/** The implementation of methods clear(), contains(Object o),
 *  get(int index), indexOf(Object o), lastIndexOf(Object o),
 *  remove(Object o), and set(int index, Object o) are omitted and  */

/** Replace the element at the specified position in this list
 *  with the specified element. */
public Object set(int index, Object o) {
  // Implementation left as an exercise
  return null;
}
```

5.4 A Java Implementation of Linked Lists

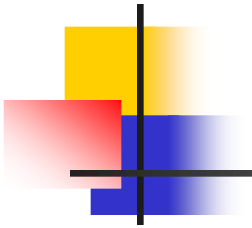


```
private class Node {
    Object element;
    Node next;

    public Node(Object o) {
      element = o;
    }
  }
}
```



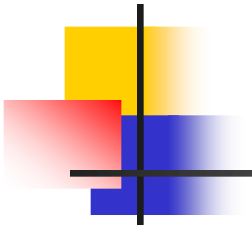
5.5 Linked Lists Application

```
public class TestLinkedList {
  public static void main(String[] args) {
    // Create a list
    MyLinkedList list = new MyLinkedList();

    // Add elements to the list
    list.addFirst("America"); // Add it to the beginning
    System.out.println("(1) " + list);
```



5.5 Linked Lists Application

```
list.addLast("Canada"); // Add it to the last of the list
System.out.println("(2) " + list);

list.addFirst("Russia"); // Add it to the first of the list
System.out.println("(3) " + list);

list.addLast("France"); // Add it to the last of the list
System.out.println("(4) " + list);
```

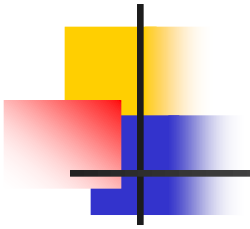


5.5 Linked Lists Application

```
list.add(2,"Germany"); // Add it to the list at index 2
System.out.println("(5) " + list);

list.add(0,"Norway"); // Add it to the list at index 0
System.out.println("(6) " + list);

list.add(4,"Netherlands"); // Same as list at index 4
System.out.println("(7) " + list);
```

5.5 Linked Lists Application

```
// Remove elements from the list
    list.remove(4); // Remove element at index 4
    System.out.println("(8) " + list);

    list.removeFirst(); // Remove the first element
    System.out.println("(9) " + list);

    list.removeLast(); // Remove the last element
    System.out.println("(10) " + list);
  }
}
```



Mind Test

Mind Test

What argument should be added into traverse function definition to print the elements in the list?

Stack using linked-based representation

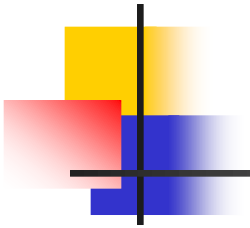
The ArrayStack implementation is inefficient because it requires rebuilding the array when it gets full:

```
public class LinkedStack implements Stack {
        private Node top;
        private int size;

        public Object peek() {
                if (size == 0) throw new java.util.NoSuchElementException();
                return top.object;
        }

        public Object pop() {
                if ( size == 0) throw new java.util.NoSuchElementException();
                Object oldTop = top.object;
                top = top.next;
                --size;
                return oldTop;
        }
```

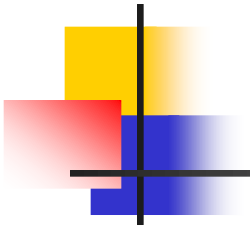
```
        public void push(Object object) {
                top = new Node(object, top);
                ++size;
        }

        public int size () {
                return size;
        }

        private static class Node {
                Object object;
                Node next;

                Node (Object object, Node next) {
                        this.object = object;
                        this.next = next;
                }
        }
}
```

Queue using linked-based representation

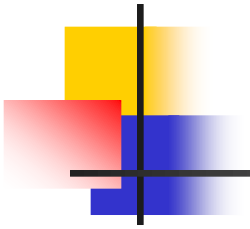
```
public class LinkedQueue implements Queue{
        private Node head = new Node(null);
        private int size;

        public Object enqueue(Object object) {
                head.prev = head.prev.next = new Node(object, head.prev, head);
                ++size;
        }

        public Object dequeue() {
                if ( size == 0) throw new IllegalStateException("the queue is empty");
                Object object = head.next.object;
                head.next= head.next.next;
                head.next.prev=head;
                --size;
                return object;
        }
```



```
public Object first() {
        if (size == 0) throw IllegalStateException("the queue is empty");
        return head.next.object;
}

public boolean isEmpty() {
    return size==0;
}

public int size() {
    return size;
}
```



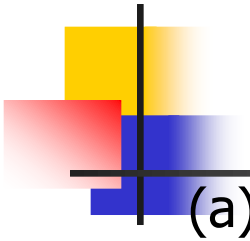
```
private static class Node {
        Object object;
        Node prev=this, next=this;

        Node(Object object) {
                this.object = object;
        }

        Node(Object object, Node prev, Node next) {
                this.object = object;
                this.prev = prev;
                this.next = next;
        }
```



Exercise

Assume the following declarations:

```
public class Node {
      int data;
      Node next;

      public Node (int element) {
            data = element;
       }
    }
```



Exercise

(a) After each program statement, display the resulting chain of nodes.

```
front = new Node (6);
newNode = new Node (13);
newNode.next = front;
front = newNode;
curr = front.next;

for (i = 3; i >=1; i--) {
  newNode = new Node (i);
  curr.next = newNode;
  curr = newNode; }
```

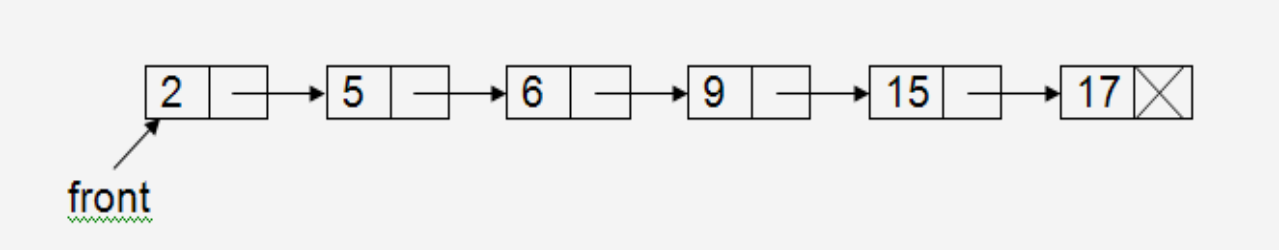
Exercise

(b) Given the following static java method to remove an element from a linked list.

```
// delete the first occurrence of the target element
// in the linked list referenced by front;
// returns the value of front

public static Node remove(Node front, int element)
{
   // curr moves through list, trailed by prev
   Node curr = front, prev = null;

   //  becomes true if we locate target
   boolean foundItem = false;

   // scan until locate item or come to end of list
   while (curr != null && !foundItem)
   {

      //check for a match; if found, check whether deletion
      //occurs at the front or at an intermediate position
      //in the list; set Boolean foundItem true

       if (element.equals(curr.data))
       {
          if (prev = = null)
              front = front.next;
          else
              prev.next = curr.next;
          foundItem = true;
       }
       else
       {
       // advance curr and prev
          prev = curr;
          curr= curr.next;
       }
   }

   //return current value of front which is updated when
the
   //deletion occurs at the first element in the list

return front;
}
```

Exercise

Based on the given method and the following linked list, display the resulting chain of nodes after these two method calls are invoked one after another. Show the positions of front, curr and prev.



i. remove(front, 6)

ii. remove(front, 2)