

Lists

A list is a finite, ordered sequence of data items called elements.

Each list element has a data type.

The empty list contains no elements.

The length of the list is the number of elements currently stored.

The beginning of the list is called the head, the end of the list is called the tail.

Sorted lists have their elements positioned in ascending order of value, while unsorted lists have no necessary relationship between element values and positions.

Notation: $(a_0, a_1, \dots, a_{n-1})$

What operations should we implement?

List ADT

```
interface List { // List ADT
public void clear(); // Remove all Objects
public void insert(Object item); // Insert at curr pos
public void append(Object item); // Insert at tail
public Object remove(); // Remove/return curr
public void setFirst(); // Set to first pos
public void next(); // Move to next pos
public void prev(); // Move to prev pos
public int length(); // Return curr length
public void setPos(int pos); // Set curr position
public void setValue(Object val); // Set current value
public Object currValue(); // Return curr value
public boolean isEmpty(); // True if empty list
public boolean isInList(); // True if in list
public void print(); // Print all elements
} // interface List
```

List ADT Examples

List: (12, 32, 15)

```
MyLst.insert(element);
```

Assume MyPos has 32 as current element:

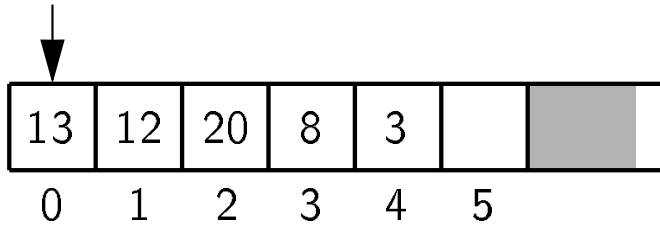
```
MyLst.insert(99);
```

Process an entire list:

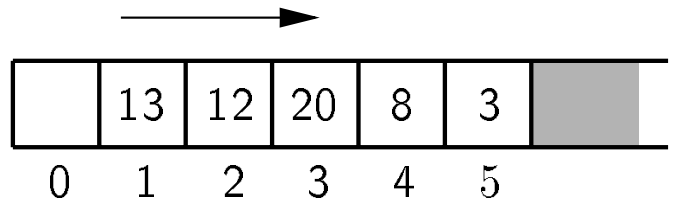
```
for (MyLst.setFirst(); MyLst.isInList(); MyLst.next())  
    DoSomething(MyLst.currValue());
```

Array-Based List Insert

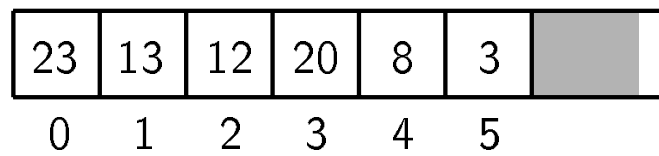
Insert 23:



(a)



(b)



(c)

Array-Based List Class

```
class AList implements List { // Array-based list

private static final int defaultSize = 10;

private int msize; // Maximum size of list
private int numInList; // Actual list size
private int curr; // Position of curr
private Object[] listArray; // Array holding list

AList() { setup(defaultSize); } // Constructor
AList(int sz) { setup(sz); } // Constructor

private void setup(int sz) { // Do initializations
    msize = sz;
    numInList = curr = 0;
    listArray = new Object[sz]; // Create listArray
}

public void clear() // Remove all Objects from list
{ numInList = curr = 0; // Simply reinitialize values

public void insert(Object it) { // Insert at curr pos
    Assert.notFalse(numInList < msize, "List is full");
    Assert.notFalse((curr >=0) && (curr <= numInList),
        "Bad value for curr");
    for (int i=numInList; i>curr; i--) // Shift up
        listArray[i] = listArray[i-1];
    listArray[curr] = it;
    numInList++; // Increment list size
}
```

Array-Based List Class (cont)

```
public void append(Object it) { // Insert at tail
    Assert.notNull(numInList < msize, "List is full");
    listArray[numInList++] = it; // Increment list size
}

public Object remove() { // Remove and return Object
    Assert.notNull(!isEmpty(), "No delete: list empty");
    Assert.notNull(isInList(), "No current element");
    Object it = listArray[curr]; // Hold removed Object
    for(int i=curr; i<numInList-1; i++) // Shift down
        listArray[i] = listArray[i+1];
    numInList--; // Decrement list size
    return it;
}

public void setFirst() { curr = 0; } // Set to first
public void prev() { curr--; } // Move curr to prev
public void next() { curr++; } // Move curr to next
public int length() { return numInList; }
public void setPos(int pos) { curr = pos; }
public boolean isEmpty() { return numInList == 0; }

public void setValue(Object it) { // Set current value
    Assert.notNull(isInList(), "No current element");
    listArray[curr] = it;
}

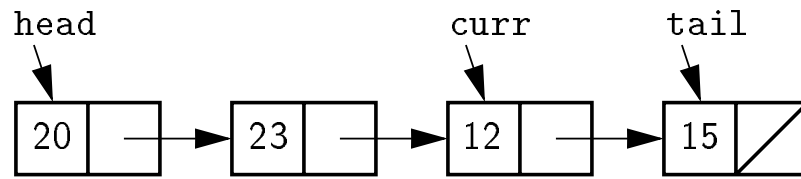
public boolean isInList() // True if curr within list
    { return (curr >= 0) && (curr < numInList); }
} // Array-based list implementation
```

Link Class

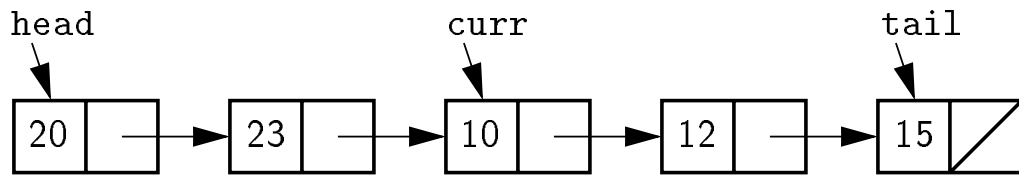
Dynamic allocation of new list elements.

```
class Link { // A singly linked list node
  private Object element; // Object for this node
  private Link next; // Pointer to next node
  Link(Object it, Link nextval) // Constructor
    { element = it; next = nextval; }
  Link(Link nextval) { next = nextval; } // Constructor
  Link next() { return next; }
  Link setNext(Link nextval) { return next = nextval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }
}
```

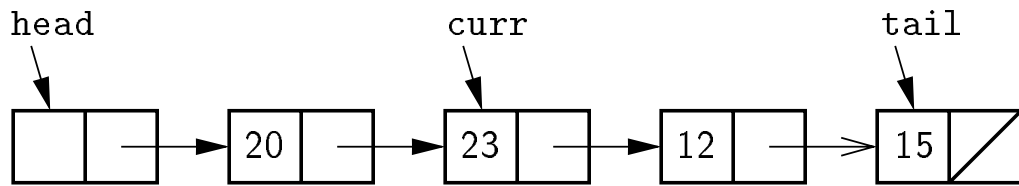
Linked List Position



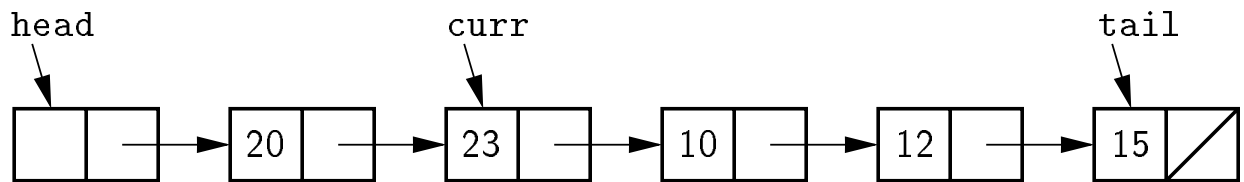
(a)



(b)



(a)



(b)

Linked List Implementation

```
public class LList implements List { // Linked list
private Link head;    // Pointer to list header
private Link tail;    // Pointer to last Object in list
protected Link curr; // Position of current Object

LList(int sz) { setup(); } // Constructor
LList() { setup(); } // Constructor
private void setup()
    { tail = head = curr = new Link(null); }

public void setFirst() { curr = head; }
public void next()
    { if (curr != null) curr = curr.next(); }

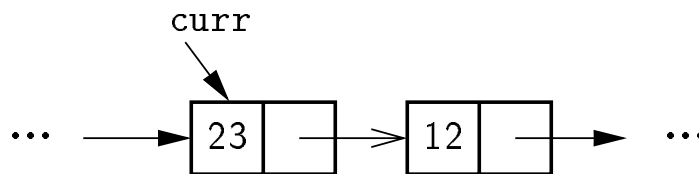
public void prev() { // Move to previous position
    Link temp = head;
    if ((curr == null) || (curr == head)) // No prev
        { curr = null; return; } // so return
    while ((temp != null) && (temp.next() != curr))
        temp = temp.next();
    curr = temp;
}

public Object currValue() { // Return current Object
    if (!isInList()) return null;
    return curr.next().element();
}

public boolean isEmpty() // True if list is empty
    { return head.next() == null; }
} // Linked list class
```

Linked List Insertion

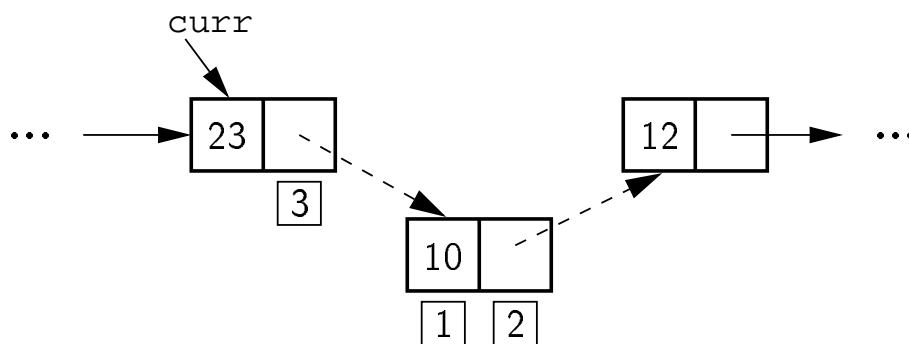
```
// Insert Object at current position
public void insert(Object it) {
    Assert.notNull(curr, "No current element");
    curr.setNext(new Link(it, curr.next()));
    if (tail == curr)           // Appended new Object
        tail = curr.next();
}
```



Insert 10:

10	
----	--

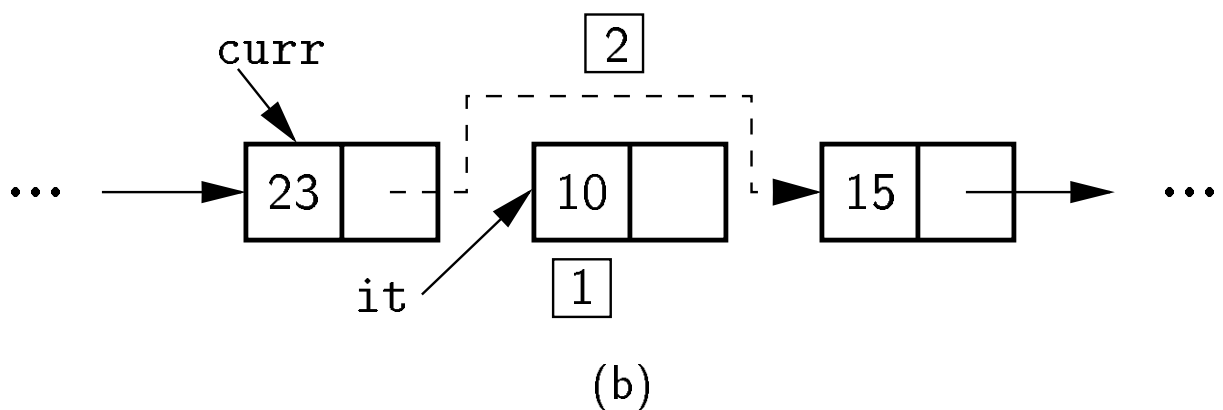
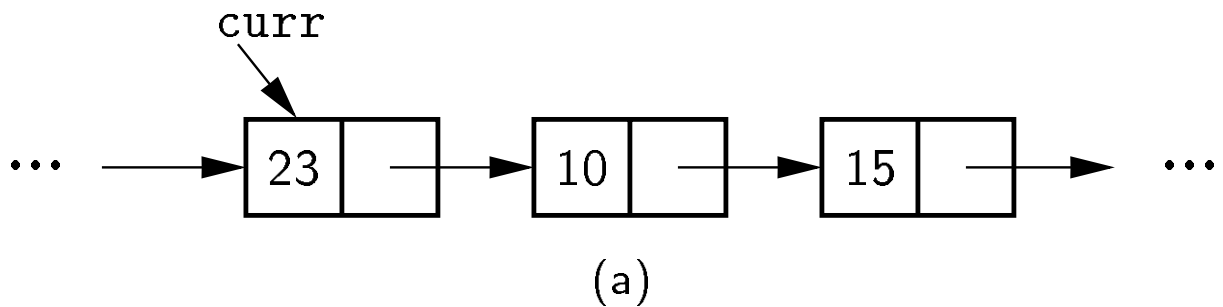
(a)



(b)

Linked List Remove

```
public Object remove() { // Remove/return curr Object
    if (!isInList()) return null;
    Object it = curr.next().element(); // Remember value
    if (tail == curr.next()) tail = curr; // Set tail
    curr.setNext(curr.next().next()); // Cut from list
    return it; // Return value
}
```



Freelists

System new and garbage collection are slow.

```
class Link { // Singly linked list node with freelist
    private Object element; // Object for this Link
    private Link next;      // Pointer to next Link
    Link(Object it, Link nextval)
    { element = it; next = nextval; }
    Link(Link nextval) { next = nextval; }
    Link next() { return next; }
    Link setNext(Link nextval) { return next = nextval; }
    Object element() { return element; }
    Object setElement(Object it) { return element = it; }

    // Extensions to support freelists
    static Link freelist = null; // Freelist for class

    static Link get(Object it, Link nextval) {
        if (freelist == null)
            return new Link(it, nextval);
        Link temp = freelist;
        freelist = freelist.next();
        temp.setElement(it);
        temp.setNext(nextval);
        return temp;
    }

    void release() {
        element = null; next = freelist; freelist = this;
    }
}
```

Comparison of List Implementations

Array-Based Lists:

- Insertion and deletion are $\Theta(n)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:

- Insertion and deletion $\Theta(1)$;
prev and direct access are $\Theta(n)$.
- Space grows with number of elements.
- Every element requires overhead.

Space “break-even” point:

$$DE = n(P + E); \quad n = \frac{DE}{P + E}$$

E: Space for data value

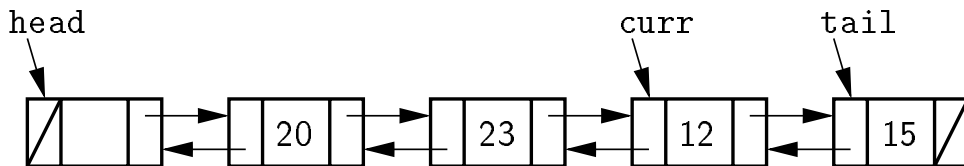
P: Space for pointer

D: Number of elements in array

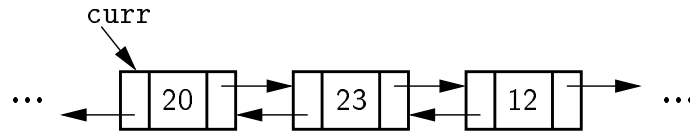
Doubly Linked Lists

Simplify insertion and deletion: Add a prev pointer.

```
class DLink { // A doubly-linked list node
  private Object element; // Object for this node
  private DLink next; // Pointer to next node
  private DLink prev; // Pointer to previous node
  DLink(Object it, DLink n, DLink p)
  { element = it; next = n; prev = p; }
  DLink(DLink n, DLink p) { next = n; prev = p; }
  DLink next() { return next; }
  DLink setNext(DLink nextval) { return next=nextval; }
  DLink prev() { return prev; }
  DLink setPrev(DLink prevval) { return prev=prevval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }
}
```



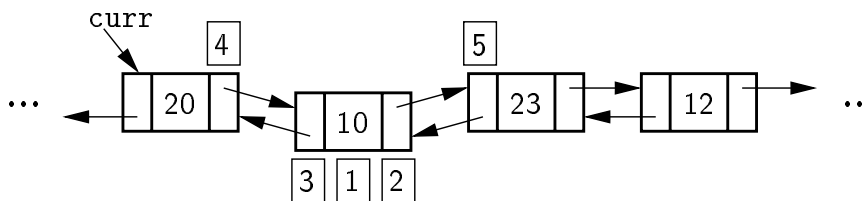
Doubly Linked List Operations



Insert 10:

	10	
--	----	--

(a)



(b)

```
// Insert Object at current position
public void insert(Object it) {
    Assert.notNull(curr, "No current element");
    curr.setNext(new DLink(it, curr.next(), curr));
    if (curr.next().next() != null)
        curr.next().next().setPrev(curr.next());
    if (tail == curr) // Appended new Object
        tail = curr.next();
}

public Object remove() { // Remove/return curr Object
    Assert.notNull(isInList(), "No current element");
    Object it = curr.next().element(); // Remember Object
    if (curr.next().next() != null)
        curr.next().next().setPrev(curr);
    else tail = curr; // Removed last Object: set tail
    curr.setNext(curr.next().next()); // Remove from list
    return it; // Return value removed
}
```

Stacks

LIFO: Last In, First Out

Restricted form of list: Insert and remove only at front of list.

Notation:

- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.

Array-Based Stack

Define top as first free position.

```
class Stack class AStack { // Array based stack class
    private static final int defaultSize = 10;
    private int size;           // Maximum size of stack
    private int top;           // Index for top Object
    private Object [] listarray; // Array holding stack
    AStack() { setup(defaultSize); }
    AStack(int sz) { setup(sz); }

    public void setup(int sz)
    { size = sz; top = 0; listarray = new Object[sz]; }

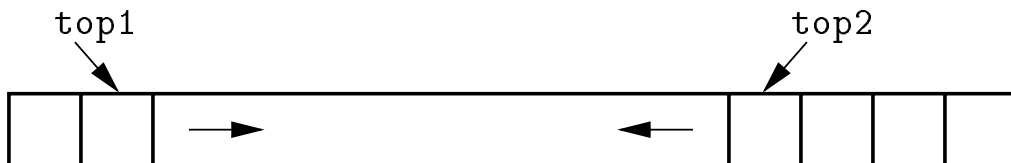
    public void clear() { top = 0; } // Clear all Objects

    public void push(Object it) // Push onto stack
    { Assert.notFalse(top < size, "Stack overflow");
      listarray[top++] = it; }

    public Object pop()           // Pop Object from top
    { Assert.notFalse(!isEmpty(), "Empty stack");
      return listarray[--top]; }

    public Object topValue()      // Return top Object
    { Assert.notFalse(!isEmpty(), "Empty stack");
      return listarray[top-1]; }

    public boolean isEmpty() { return top == 0; }
};
```



Linked Stack

```
public class LStack { // Linked stack class

private Link top; // Pointer to list header

public LStack() { setup(); } // Constructor
public LStack(int sz) { setup(); } // Constructor

private void setup() // Initialize stack
{ top = null; } // Create header node

public void clear() { top = null; } // Clear stack

public void push(Object it) // Push Object onto stack
{ top = new Link(it, top); }

public Object pop() { // Pop Object from top
    Assert.assertFalse(!isEmpty(), "Empty stack");
    Object it = top.element();
    top = top.next();
    return it;
}

public Object topValue() // Get value of top Object
{ Assert.assertFalse(!isEmpty(), "No top value");
  return top.element(); }

public boolean isEmpty() // True if stack is empty
{ return top == null; }
} // Linked stack class
```

Queues

FIFO: First In, First Out

Restricted form of list:

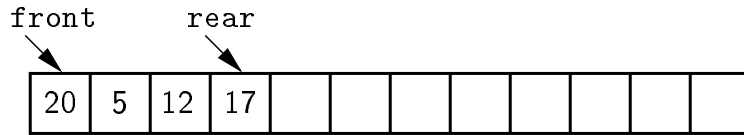
Insert at one end, remove from other.

Notation:

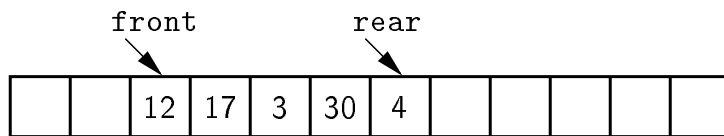
- Insert: Enqueue
- Delete: Dequeue
- First element: FRONT
- Last element: REAR

Queue Implementations

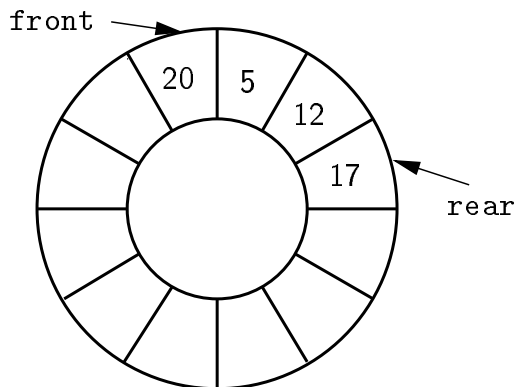
Array-Based Queue



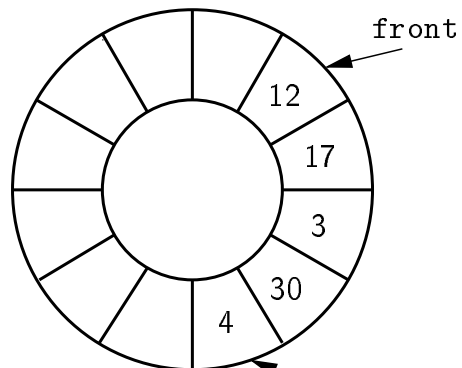
(a)



(b)



(a)



(b)

Linked Queue: modified linked list.