**Why linked list? (A: collections)**

**Why not arrays? (A: differences between arrays and linked lists)**

|  |  |
| --- | --- |
| Arrays | Linked List |
| Fast access to any element because of direct access | Slower access to any element because it has to be searched sequentially |
| Access objects directly without casting | Needs casting during retrieval |
| Controlled size | Auto maintains order |
| Simpler to program | Complex to program |
| Fixed in size, can’t resize at runtime, it has to be known at compile time | Resize dynamically at runtime |
| Complex maintenance or organized | Auto maintenance, self-organizing |
| Consist of contagious (continuous) memories | Consist of nodes, each pointing to the next node that is located anywhere in memory |
| Each element has fixed size in bytes | Each node can be a different size |
| Require less memory | Requires more memory because of the addition of pointers |
| Deleting an element in the array is harder | Deleting a node is easy |
| Inserting an element in the array is difficult | Inserting a node is easy |
| Allows random access using the index | Does not allow random access |
|  | Can be circular, or doubly-linked (backward too) |
|  | To find any node, each previous node must be examined and pointers followed. |
| If immediate access of any element is necessary, then the array is a better choice, e.g. binary search, and most sorting algorithms | If constantly accessing only some elements—the first, the second, the last, and the like—and if changing the structure is the core of an algorithm, then linked list is a better option |

For large linked lists, a significant amount of memory is needed to store the pointers. Therefore, if a problem does not require many shifts of data, then having an oversized array may not be wasteful at all if its size is compared to the amount of space needed for the linked structure storing the same data as the array.

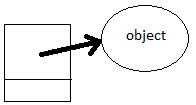
**Singly Linked List:**

Two types of nodes:

1. Nodes that contain primitive info, example:

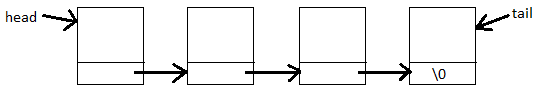


1. And Nodes that point to object type info, example:

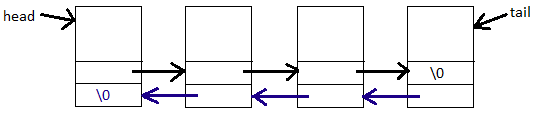


:

If these nodes are strung together, you can create a singly linked list

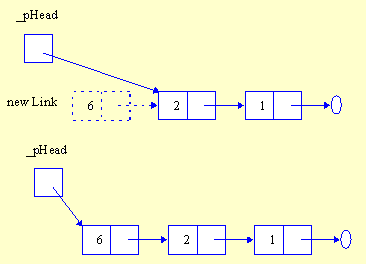


Or, doubly linked list, not part of the lecture, but you're encourage to explore!

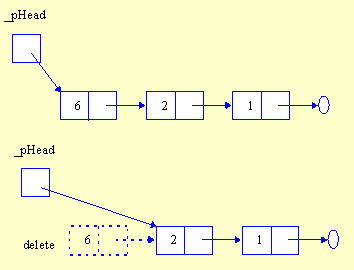


**Linked List**

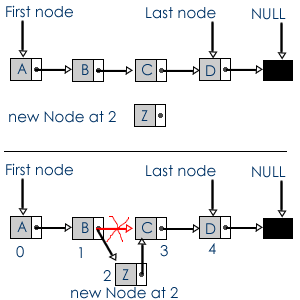
Adding a node:



Deleting node in a link list:



Inserting a node:



API:

public interface List<E> {

/\*\*

\* Remove all contents from the list, so it is once again empty. Client is

\* responsible for reclaiming storage used by the list elements.

\*/

public void clear();

/\*\*

\* Insert an element at the current location. The client must ensure that

\* the list’s capacity is not exceeded.

\*

\* @param item The element to be inserted.

\*/

public void insert(E item);

/\*\*

\* Append an element at the end of the list. The client must ensure that the

\* list’s capacity is not exceeded.

\*

\* @param item The element to be appended.

\*/

public void append(E item);

/\*\*

\* Remove and return the current element.

\*

\* @return The element that was removed.

\*/

public E remove();

/\*\*

\* Set the current position to the start of the list

\*/

public void moveToStart();

/\*\*

\* Set the current position to the end of the list

\*/

public void moveToEnd();

/\*\*

\* Move the current position one step left. No change if already at

\* beginning.

\*/

public void prev();

/\*\*

\* Move the current position one step right. No change if already at end.

\*/

public void next();

/\*\*

\* @return The number of elements in the list.

\*/

public int length();

/\*\*

\* @return The position of the current element.

\*/

public int currPos();

/\*\*

\* Set current position.

\*

\* @param pos The position to make current.

\*/

public void moveToPos(int pos);

/\*\*

\* @return The current element.

\*/

public E getValue();

}