# **The List ADT**

* We will deal with a general list of the form *A*0, *A*1, *A*2, ..., *AN*−1.
* We say that the size of this list is *N*.
* We will call the special list of size 0 an **empty list**.
* For any list except the empty list, we say that
  + *Ai* follows (or succeeds) *Ai*−1 (*i* < *N*)
  + and
  + *Ai*−1 precedes *Ai* (*i* > 0)
* The first element of the list is *A*0, and the last element is *AN*−1.
* We will not define
  + the predecessor of *A*0
  + or
  + the successor of *AN* −1.
* The **position** of element *Ai* in a list is *i*.
* Throughout this discussion, we will assume, to simplify matters, that the elements in the list are integers
* In general, arbitrarily complex elements are allowed (and easily handled by a generic Java class).

**List<E> Common Operations**

* Associated with these “definitions” is a set of **operations** that we would like to perform on the list ADT.
* Some common operations are
  + **toString**() – prints the list
  + **get**(int index) – returns the element at the specified index
  + **set**(int index) – sets the element at the specified index
  + **findKth**(kth element) – returns the kth element
  + **add**(E e) – add an element at last position
  + **add**(int index, E e) – add an element to position index
  + **remove**(int index) – remove an element at position

# **ArrayList Implementation**

**ArrayList<E>** – Provides a growable array implementation of the List ADT.

## **Operation Runtime**

* **toString**() – carried out in linear time O(N).
* **get**(int index) – carried out in constant time O(1).
* **set**(int index) – carried out in constant time O(1).
* **findKth**(kth element) – operation takes constant time O(1).
* **add**(E e) – carried out in amortized constant time O(1).
* **add**(int index, E e) – either O(N) or O(1) (**potentially expensive)**
* **remove**(int index) – either O(N) or O(1) (**potentially expensive)**
* The runtime of **add**(int index, E e) and **remove**(int index) dependson ***where*** the insertions and deletions occur.

1. **Worst case:** *O*(*N*) if we add or remove the **first element** (beginning/front of list)
   1. **add**(E e, int index) – inserting into position 0 requires pushing the entire array right one spot to make room.
   2. **remove**(int index) – deleting the first element requires shifting all the elements in the list left one spot.
2. **Average Case:** *O*(*N*) if we add or remove in **middle of list**
   1. add(E e, int index) – on average, half of the list must be moved right. O(N/2) is still equal to O(N) if we drop the constants, so linear time is still required.
   2. remove(int index) – on average, half of the list must be moved left. O(N/2) is still equal to O(N) if we drop the constants, so linear time is still required.
3. **Best Case:** *O*(*1*) if we add or remove at **end of list**
   1. add(E e, int index) – no elements need to be shifted.
   2. remove(int index) – no elements need to be shifted.

## **Tradeoffs**

* **Advantages:**
  + get(int index) takes constant time O(1)
  + set(int index, AnyType x) takes constant time O(1)
  + add() and remove() are O(1) if you are inserting/removing at the end of a list.
* An array implementation is suitable if a list is
  + built up by insertions at the high end
  + have rare deletions

and

* + repetitive/common array accesses (i.e., findKth operations)
* **Disadvantages:**
  + add() and remove() are O(N) if you are inserting/removing at the beginning or middle of a list because you must shift all succeeding values in the array down 1 index.
* An array implementation is not suitable if insertions/deletions/gets/sets occur in the middle of the list, and in particular, at the front of the list.
* The next subsection deals with the alternative: the *linked list.*

# **Linked List Implementation**

**LinkedList<E>** – Provides a doubly linked list implementation of the List ADT.

* Our main runtime issue for **add**(int index, E e) and **remove**(int index) were inserting or deleting at the **head of a list** in **linear time** O(N).
* In order to avoid the linear cost of insertion and deletion at the **head of a list**, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved.
* A linked list solves this issue for use, because a linked list consists of a series of **nodes**, which are ***not necessarily adjacent in memory***.

## **Operation Runtime**

* **toString**() – carried out in linear time O(N).
* **get**(int index) – O(1) if getting head or tail, all others nodes are O(N).
* **set**(int index) – O(1) if getting head or tail, all others nodes are O(N).
* **findKth**(kth element) – operation takes constant time O(i).
* With toString(), get(int index), and add(E e), the constant is likely to be larger with a linked list compared to an array implementation in both time and space.
* The get(int index) and set(int index, E e) operations now take linear time to complete, because we cannot index a linked list like we can an array.
* The findKth(kth element) operation is no longer quite as efficient as an array implementation.
  + We start at the first node in the list and then traverse the list until we find the kth element because we cannot index the linked list like we can an array.
  + In practice, this bound is pessimistic, because frequently the calls to findKth are in sorted order (by *i*).
  + As an example, findKth(2), findKth(3), findKth(4), and findKth(6) can all be executed in one scan down the list.
* **add(int index, E e) – O(N) (with special cases)**

The insert method requires obtaining a new node from the system by using a new call and then executing two reference maneuvers.

The general idea is shown in Figure 3.3. The dashed line represents the old next reference.

Diagram

Description automatically generated

1. **Worst case:** *O*(*N*) if inserting in middle of list
2. **Average Case:** *O*(*N*) if inserting in middle of list
3. **Best Case:** *O*(*1*) if inserting to the head or tail of the list

**Adding To The Front of the list O(1)**

The special case of adding to the front is a constant-time operation, presuming of course that a link to the front of the linked list is maintained.

In principle, if we know where a change is to be made, inserting an item into a linked list does not require moving lots of items and instead involves only a **constant** number of changes to node links.

Thus, a typical linked list **keeps links to both ends of the list (a head and tail pointer)**.

**Adding To The End of the List O(1)**

The special case of adding at the end (i.e., making the new item as the last item) can be constant-time, as long as we maintain a link to the last node.

* **remove(int index) – O(*N*) (with special cases)**
  + The remove method can be executed in one **next** reference change.
  + The to-be-deleted node’s predecessor’s next link is set to the to-be-deleted node’s next link.
  + Figure 3.2 shows the result of deleting the third element in the original list.

Diagram

Description automatically generated

**Removing The Front of the List O(1)**

The special case of removing the first item is a constant-time O(1) operation, **IF** there is a link to the front of the linked list is maintained.

This will be our head pointer, so removing from the front of a list should take O(1) time.

In principle, if we know where a change is to be made, removing an item from a linked list does not require moving lots of items and instead involves only a **constant** number of changes to node links.

**Removing The End of the List O(1)**

Removing the last item is trickier, because we have to find the next-to-last item, change its *next* link to null, and then update the link that maintains the last node.

This would result in an O(n-1) runtime every time we deleted an item from the end of the list.

In the classic linked list, where each node stores a link to its next node, having a link to the last node provides no information about the next-to-last node.

**Solution:**

The obvious idea of maintaining a third link to the next-to-last node doesn’t work, because it too would need to be updated during a remove.

Instead, we have every node maintain a link to its previous node in the list, while also maintain links to both ends of the list.

This is shown in Figure 3.4 and is known as a **doubly linked list.**

Diagram

Description automatically generated

## **Tradeoffs**

* **Advantages:** 
  + add(), remove(), get(), and set() are O(1) if the position node operated on is at the tail or head of the list.
  + This means that these operations are **constant-time** operations **only if** we maintain **links to the front and back of the list** (head and tail pointers) in a doubly linked list.
* Java’s LinkedList class takes advantage of this fact by providing methods such as
  + addFirst(E e) and addLast(E e)
  + removeFirst() and removeLast()
  + getFirst() and getLast()
* to efficiently add, remove, and access the items at both ends of the list.
* Therefore, if you have an application that frequently accesses the head and tail of a collection, and accesses the middle portions less frequently, use a linked list.
* **Disadvantages:** 
  + add() and remove() are O(N) if you are inserting/removing in the middle of a list.
  + get(int index) and set(int index, AnyType x) are O(N) unless the index is at the head or tail of the list.
* The runtimes are access to the middle of a list are linear because a LinkedList is **not easily indexable** **like an array**, and traversal must be done from either the beginning or end of the list.
  + If the call to get is for an item near the **back** of the list, the search can proceed from the back of the list.
  + If the call to to get is for an item near the **front** of the list, the search can proceed from the front of the list.

# **ArrayList vs. LinkedList Runtime**

* To see the differences, we look at some methods that operate on a List.

**Adding items to the end:**

* Suppose we construct a List by adding items at the end.

Text, letter

Description automatically generated

* LinkedList runtime = *O*(*N*)
* ArrayList runtime = *O*(*N*)
* Regardless of whether an ArrayList<E> or LinkedList<E> is passed as a parameter, the running time of makeList1 is *O*(*N*).
* We run through all N items in the list which takes *O*(*N*) runtime.
* Each call to add() appends items to the end of the list, which takes *O*(*1*) constant time (the occasional expansion of the ArrayList amortized, so it is safe to ignore).

**Adding items to the front:**

* Now let’s construct a List by adding items at the front.

Text, letter

Description automatically generated

* LinkedList runtime = *O*(*N*)
* ArrayList runtime = *O*(*N*2)
* We run through all N items in the list which takes *O*(*N*) runtime.
* Each call to add(int index, E e) adds a value to the front of the list
  + For a LinkedList, this only required moving pointers, which takes *O*(*1*) constant time.
  + For an ArrayList, adding at the front requires to move all other elements down an index, which is an *O*(*N*) operation.

**Getting (indexing) items**

* The next routine attempts to compute the sum of the numbers in a List:

Text

Description automatically generated

* LinkedList runtime = *O*(*N*2)
* ArrayList runtime = *O*(*N*)
* We run through all N items in the list which takes *O*(*N*) runtime.
* Each call to get(int index) queries for a value at each index in the list.
  + For a LinkedList, because in a LinkedList, calls to get are *O*(*N*) operations because a they are not easily indexable.
  + For an ArrayList, the underlying data structure is an array, which is easily indexable in *O*(*1*) constant time.
* **If we wanted to reduce the runtime for a LinkedList from *O*(*N2*) to *O*(*N*) runtime, 2e could instead use an enhanced for loop, which will make the running time *O*(*N*) for any List, because the iterator will efficiently advance from one item to the next.**
* Both ArrayList and LinkedList are inefficient for searches, so calls to the Collection contains(AnyType E) method and remove(AnyType E) methods (that take an AnyType as parameter) take linear time O(N).