# 1
## Index-Based Lists

## 1.1 Abstract Data Types and Structures

- An abstract data type (ADT) is a specification of the desired behavior from the point of view of the user of the data, leaving the implementation details to the programmer.

- A data structure is a concrete representation of data that is from the point of view of an implementer, not a user.

## 1.2 Index-Based List methods

| Method | Description |
|--------|-------------|
| size() | (int) number of elements in the store |
| isEmpty() | (boolean) whether the list is empty |
| get(i) | return element at index i |
| set(i,e) | replace the element at i with e and return the replaced element |
| add(i,e) | insert element e at index i and shift up elements with indexes >= i |
| remove(i) | remove and return the element at index i and shift down indexes >= i |

## 1.3 Array-based List

Time complexity of get(i) and set(i,e) is O(1) time, as they are independent of the size of the array (N) or the represented list (n) as an array-based implimentation allows for random access.

```
1. def get(i: int):
2.     if i < 0 or i >= n.size() then
3.         return "Index out of bound"
4.     else
5.         return A[i]
```

```
1. def set(i: int, e: element):
2.     if i < 0 or i >= n.size() then
3.         return "Index out of bound"
4.     else
5.         result = A[i]
6.         A[i] = e
7.         return result
```

In an operation add(i, e), we must make room for the new element by shifting forward $n - i$ elements A[i], ..., A[$n - 1$]. Time complexity is O(n) in the worst case that i = 0. The same principles apply for remove(i).

```
1. def add(i: int, e: element):
2.     if n = N then
3.         return "Array is full"
4.     if i < n then
5.         for j in [n-1, n-2, ..., i] do
6.             A[j+1] ← A[j]
7.     A[i] ← e
8.     n ← n + 1
```

```
1. def remove(i: int):
2.     if i < 0 or i >= n then
3.         return "Index out of bound"
4.     e ← A[i]
5.     if i < n − 1 then
6.         for j in [i+1, i+2, ..., n-1] do
7.             A[j] ← A[j+1]
8.     n ← n - 1
9.     return e
```

# 2
# Positional lists

ADT for a list where we store elements at "positions." Position models the abstract notion of place where a single object is stored within a container data structure. Unlike index, this keeps referring to the same entry even after insertion/deletion happens elsewhere in the collection.

## 2.1 Positional List methods

| Method | Description |
|---|---|
| size() | (int) number of elements in the store |
| isEmpty() | (boolean) whether the list is empty |
| first() | returns the position of first element (null if empty) |
| last() | returns the position of last element (null if empty) |
| before(p) | return the position immediately before p (null if p is first) |
| after(p) | returns the position immediately after p (null if p last) |
| insertBefore(p, e) | inserts e in front of the element at position p |
| insertAfter(p, e) | inserts e following the element at position p |
| remove(p) | remove and return the element at position p |

## 2.2 Singly linked Lists

A singly linked list is a concrete data structure that consistes of a sequence of nodes. The list is captured by a reference to the first node of the list, known as the head. Each node in a singly linked lists stores its element and a link to the next node. The last node stores a null reference instead of a link.

```
1. def insertFirst(e: element):                                    # O(1)
2.      Instaniate a new node x
3.      Set e as element of x
4.      Set x.next to point to head
5.      Update head to point to x
```

```
1. def removeFirst(e: element):                                    # O(1)
2.      Update head to point to next node
3.      Delete the former first node
```

```
1. def insertBefore(p, e):                                         # O(n)
2.      To find the predecessor of p we need to follow the links from the "head".
```

## 2.3 Doubly linked lists

A very natural way to implement a positional list is with a doublylinked list, so that it is easy/quick to find the position before. Each Node in a Doubly Linked List stores its element, and a link to the previous and next nodes.

```
1. def insertBefore(p, e):                                         # O(1)
2.      Instaniate a new node x
3.      Set e as element of x
4.      Set x.next to point to p
5.      Set x.prev to point to p.prev
6.      Set p.prev.next to point to x
7.      Set p.prev to point to x
8.      return x
```

```
1. def remove(p):                                                  # O(1)
2.      Set p.prev.next to point to p.next
3.      Set p.next.prev to point to p.prev
4.      return p.element
```

| Method | Description | Time |
|---|---|---|
| size() | (int) number of elements in the store | O(1) |
| isEmpty() | (boolean) whether the list is empty | O(1) |
| first() | returns p of first element (null if empty) | O(1) |
| last() | returns p of last element (null if empty) | O(1) |
| before(p) | return the position before p (null if p is first) | O(1) |
| after(p) | returns the position after p (null if p last) | O(1) |
| insertBefore(p, e) | inserts e in front of the element at position p | O(1) |
| insertAfter(p, e) | inserts e following the element at position p | O(1) |
| remove(p) | remove and return the element at position p | O(1) |

## 2.4 Array or Linked List implementation

**Linked Lists**

- good match to positional ADT

- efficient insertion and deletion

- simpler behaviour as collection grows

- modifications can be made as collection iterated over

- space not wasted by list not having maximum capacity

**Arrays**

- good match to index-based ADT

- caching makes traversal fast in practice

- no extra memory needed to store pointers

- allow random access (retrieve element by index)

## 3

## Stacks and queues

These ADTs are forms of List with restricted insertiosn and removals

- stacks follow last-in-first-out (LIFO).

- queues follows first-in-first-out (FIFO).

## 3.1 Stacks

- push(e): inserts an element, e
- pop(): removes and returns the last inserted element
- top(): returns the last inserted element without removing it
- size(): returns the number of elements in the stack
- isEmpty(): indicates whether no elements are stored

**Applications:**

- Keep track of a history that allows undoing such as Web browser history or undo sequence in a text editor.
- Chain of method calls in a language supporting recursion
- Context-free grammars
- Auxiliary data structure for algorithms
- Component of other data structures

**Array based implementation**
Array has capacity N. Add elements from left to right. A variable t keeps track of the index of the top element. space used is O(N). Each operation takes O(1) time.

```
1. def push(e: element):                              # O(1)
2.     if t = N - 1 then
3.         return "Stack is full"
4.     else
5.         t ← t + 1
6.         S[t] ← e
```

```
1. def pop():                                         # O(1)
2.     if isEmpty() then
3.         return "Stack is empty"
4.     else
5.         t ← t -1
6.         return S[t+1]
```

```
1. def size():
2.     return t+1
```

## 3.2 Queues

- enqueue(e): inserts an element, e, at the end of the queue
- dequeue(): removes and returns element at the front of the queue
- first(): returns the element at the front without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

**Applications:**

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming
- Auxiliary data structure for algorithms
- Component of other data structures

**Array based implementation**
Array has capacity N. Space used is O(N). Each operation takes O(1) time. Can wrap around an array. start: index of the front element. end: index past the last element. size: number of stored elements. end = (start + size) mod N, so we only need to keep track of start and size.

```
1. def enqueue(e: element):                          # O(1)
2.     if size() = N then
3.         return "Queue is full"
4.     else
5.         end  ← (start + size()) mod N
6.         Q[end]  ← e
7.         size  ← size + 1
```

```
1. def dequeue():                                     # O(1)
2.     if isEmpty() then
3.         return "Queue is empty"
4.     else
5.         e  ← Q[start]
6.         start  ← (start + 1) mod N
7.         size  ← size - 1
8.         return e
```