

Class 2

Summary

Lists are simple, however, creating a good list can be difficult. One way to create a list is to use the first few elements of an **array**. A more popular method is to use a **dynamic data structure** that utilizes **pointers** to link many objects of a fixed size.

Implementations of Lists

1. A simple **partially-filled array** scheme

- Store a list of size n in an array of size $c \geq n$ with list elements $L_0 \dots L_{n-1}$ in $a[0] \dots a[n-1]$.

```
a = [1,3,7,3,2,6]
n = 3 // size of list
c = 6 // capacity of array
// represents:
L = [1,3,7]
```

Operations:

```
empty(L); // -> return(n == 0);
first(L); // -> return a[0]
rest(L); // 1) copy a[1]...a[n-1] to new array
          // 2) move a[1]...a[n-1] to a[n-2]
add(6, L); // 1) copy a[0]...a[n-1] to a new array (with 6 in the zero
          // location
          // 2) move a[0]...a[n-1] to a[1]...a[n] and store 6 in a[0]
```

Arrays

a particular sort of function, with a special implementation

Consider a 1-d array A of type S and size n .

Mathematically: A is a function $A: [n] \rightarrow S$

Implementation: A is a fixed size contiguous sequence of memory locations, each of the size needed to store a value of type S .

- Easy to access the i th element is easy
- Changing the size is not possible
- Cannot "insert" or "delete" elements

Dynamic Data Structures

- DDSs change size (eg memory consumed) as the amount of data stored in them increases/decreases.
- constructed from collections of fixed-sized objects linked together by pointers
- The number of these fixed sized objects changes

Simplist - Linked List:

eg for L = [1,3,2]

Each contain a val and a next