

Linked Lists

Suppose we have a long list of integers

list
position *items*

| | |
|---|----|
| 1 | 3 |
| 2 | 14 |
| 3 | 7 |
| 4 | 8 |
| 5 | 23 |
| 6 | 5 |
| . | . |
| . | . |
| . | . |

It is natural for us to think of storing these items in a 1-dimensional array:

(On our systems, each integer is stored in a 4-byte chunk of memory)

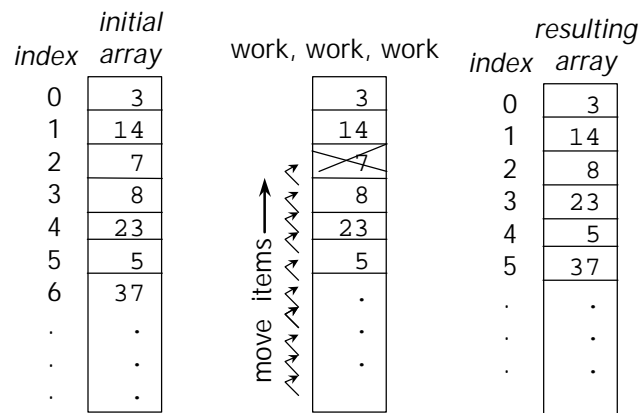
array
index *elements*

| | |
|---|----|
| 0 | 3 |
| 1 | 14 |
| 2 | 7 |
| 3 | 8 |
| 4 | 23 |
| 5 | 5 |
| . | . |
| . | . |
| . | . |

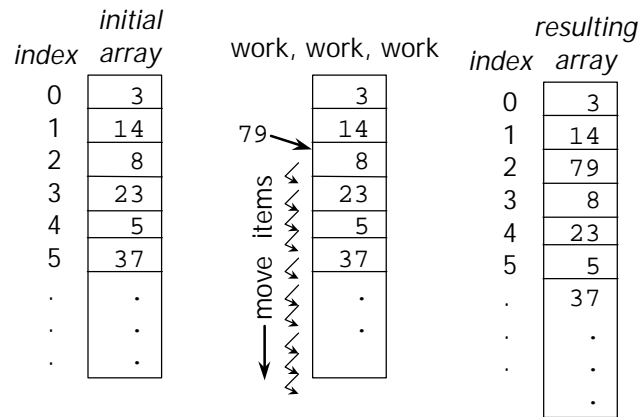
- ◆ Array elements (in C/C++) are stored next to one another in memory
- ◆ An array element's memory location is found by doing arithmetic with the array index and the start of the array
- ◆ `int MyArray[200];` *MyArray is a pointer.*
`int i, j;`
`. . .`
`j = MyArray[i];` ← This refers to the integer stored at location $(MyArray + 4*i)$

An array may seem natural for storing a list, but it brings some problems

Suppose we need to delete the item "7" from our list:



Now suppose we need to insert "79" near the front of this list:



Another problem:

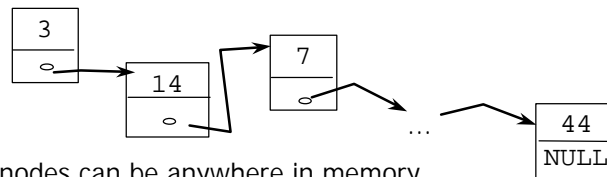
- ◆ Suppose this array is for storing a list:

```
int MyArray[200];
```

- ◆ When the program runs,
 - if there are only a few list items, then we have been wasteful with memory allocation
 - but if the list has 201 or more items, then our program cannot accommodate the list
- ◆ Instead of compile-time allocation, run-time memory allocation might be better

A **linked list** is an alternative to an array

- ◆ A linked list is a sequence of nodes, in which
 - » each node contains a data element, and
 - » each node (except the last) links to a successor node



These nodes can be anywhere in memory.
We will use pointers to link the nodes.
By convention, the `NULL` pointer indicates the end of the linked list.

Implementing linked lists in C++

- ◆ It makes sense to use C++ pointers to link the nodes in a linked list
- ◆ For one node, we'll use a `struct` to group together a data item and a pointer
- ◆ We'll use `new` and `delete` to allocate/deallocate the memory for each node at run-time, as needed

```

◆ struct listNode;
typedef listNode
*ptrType;

struct listNode
{
    int      Item;
    ptrType  Next;
};
◆ Now we could write
ptrType  P;
P = new listNode;
P->Item = 3;
P->Next = NULL;
delete P;

```

This introduces a linked-list node, to be defined later; **no memory is allocated**

This describes the components of the struct; **no memory is allocated**

allocate memory for a pointer

allocate memory for a node

deallocate node's memory

Summary: implementing linked lists in C++

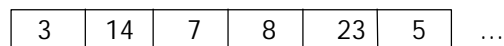
- ◆ Use a `struct` to represent a node in the linked list
- ◆ One field in the `struct` will contain the data for a particular node (this could also be a `struct`)
- ◆ One field in the `struct` will be a pointer to the next node
- ◆ At the last node of the list, store `NULL` as the pointer (to indicate the end of the list)

What operations do we need for linked lists?

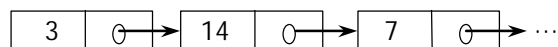
- ◆ a way to allocate memory for a new node (we just did this!), and a way to deallocate memory for a node that is no longer needed
- ◆ node insertion
- ◆ node deletion
- ◆ (possibly) special consideration of first or last nodes
- ◆ a way to progress through the linked list

Review

- ◆ We have studied arrays



and linked lists



- ◆ These are examples of low-level data structures
- ◆ We can use them to implement higher-level *Abstract Data Types*

- ◆ This will allocate memory for a node:

```
ptrType P;
```

```
P = new listNode;
```

- ◆ This will deallocate the memory:

```
delete P;
```

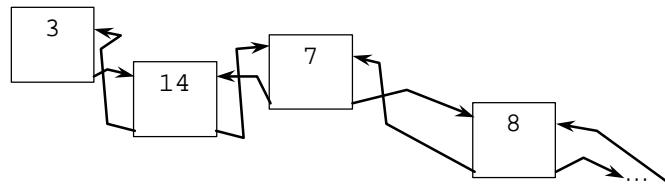
(Every chunk of memory that is obtained by new should be deleted before exiting the program.)

Variations on the linked-list concept

- ◆ **Singly linked lists** (we just studied this)
- ◆ **Doubly linked lists**
(also called two-way linked lists, or symmetrically linked lists)
- ◆ **Circular linked lists**
(can be singly or doubly linked)

Doubly linked lists

- ◆ Each node (except the first and last) links to its successor and predecessor nodes



- Need to add another pointer, for a doubly linked list

```
struct listNode;
typedef listNode *ptrType;

struct listNode
{
    int      Item;
    ptrType Next;
    ptrType Previous;
};
```

Summary

- ◆ Arrays and linked lists are examples of low-level data structures
- ◆ Many ADTs can be implemented using either arrays or linked lists
- ◆ The concept of linked lists is fairly simple
- ◆ In C/C++, the “only” problem is to keep track of pointers and memory