List is an Abstract Data Type

A list contains a sequential collection of values.

We denote the contents of a list as items, entries, or elements.

There are many different kinds of lists, but in general, we may expect a list to support operations such as:

- **Add**: add an item to the end
- **Insert**: add an item between two existing elements
- **Get**: get the value of an item at the specified index
- **Erase**: remove an item from the list
- **Size**: obtain the number of elements in the list

*Note: sequential ≠ sorted!

Array List (aka Vector)

- size = 4
- capacity = 10
- Resizes dynamically via re-allocation of array, copying elements.

Linked Lists

- Like the example array list on previous slide, this list contains (14, 36, 42, 9)

Node Class

Here’s a very simple implementation of a node:

```cpp
class node {
    public:
        int data;
        node* next;
};
```

Nodes Live on the Heap

Generally, we only keep pointers to nodes...

- Program/data structure keeps pointer to head
- Nodes keep pointers to successor nodes

Just as with array list, we want dynamic # of nodes:

- Linked list can grow/shrink
- Difference: individual nodes allocated independently
Navigating the Linked List

Iterating on linked lists:
• No memory contiguity (nodes are scattered)
• No random access (like in array)

Code to find an element:
```cpp
bool find(node* head, int val) {
    for (node* p = head; p != NULL; p = p->next) {
        if (p->data == val) return true;
    }
    return false;
}
```

Linked List Operations: add

```cpp
// start from head, travel down links to find tail.
node* ptr = head;
while (ptr->next != NULL) {
    ptr = ptr->next;
}
// ptr now points to tail node
ptr->next = new node;
ptr->next->data = 17;
ptr->next->next = NULL;
```

Linked List Operations: insert

```cpp
// assume before points to node before insert position
node* ptr = new node;
ptr->data = 7;
ptr->next = before->next;
before->next = ptr;
```

Linked List Operations: erase

```cpp
// Relink left node to right node
// Delete removed node
```
Linked List Operations: erase

```c
// assume before points to node before
// node to erase
node* ptr = before->next;
before->next = ptr->next;
delete ptr;
```

What is cost of erase?

Encapsulating Linked List

Can just keep head node, and free functions; some operations are easier/more efficient:
- Iterating over list
- Inserting/erasing elements

Disadvantages:
- User has to keep track of head/tail pointers
- User can mess up list structure with access to node internals
- No good way to keep metadata (e.g., size)
- Overall, poor encapsulation

Applications

- Very efficient operations at ends
  - Efficient insert/erase at head (Stacks)
  - Efficient add (if tail pointer), erase at head (Queues)
- Very efficient operations in middle, when pointers are kept
  - E.g., text editor (cursor acts as pointer)

What algorithms have we seen that would not be efficient on a linked list?

Efficiency: Array vs Linked

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1)</td>
<td>O(1)*</td>
</tr>
<tr>
<td>Insert</td>
<td>O(N)</td>
<td>O(1)†</td>
</tr>
<tr>
<td>Erase</td>
<td>O(N)</td>
<td>O(1)†</td>
</tr>
<tr>
<td>Indexed Get/Set</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Append</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

*With tail pointer
†At head or with pointer at location

Linked Lists and Recursion

Linked list represented as head pointer:
- Then any node* is head of a linked list
- head->next is head of a smaller linked list

Two versions of print_list():

```c
void print_list(node* head) {
    if (head == NULL) return;
    cout << head->data << endl;
    print_list(head->next);
}
```