Array List (aka Vector)

Resizes dynamically via re-allocation of array, copying elements.
- What is cost of add()?
- What is cost of insert()/erase()?

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;
    }
```

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 != nullptr; p = p->next) {
        if (p->data == val) return true;
    }
    return false;
}
```

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
Linked List Operations: add

add:
- Create new node (with next set to nullptr)
- Attach to tail

```
head
14 26 42 9 17 NULL
```

// start from head, travel down links to find tail. node *ptr = head;
while (ptr->next != nullptr)
    ptr = ptr->next;

// ptr now points to tail node
ptr->next = new node;
ptr->next->data = 17;
ptr->next->next = nullptr;

If we keep a tail pointer (in addition to head pointer), can skip first step; what is cost of add in each case?

Linked List Operations: insert

insert:
- Create new node pointing to right node
- Relink left node to new node

```
head
14 26 42 9 17 NULL
```

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

What is cost of insert?

Linked List Operations: erase

erase:
- Relink left node to right node
- Delete removed node

```
head
14 26 42 9 17 NULL
```

// 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>
<tr>
<td>...</td>
<td>Think about other operations you might use!</td>
<td></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) {
    for (node* p = head; p != NULL; p = p->next) {
        cout << p->data << endl;
    }
}
```

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

Up Next

- Friday, April 6
  - Lab 10 – Inheritance
  - Extra Credit APT due
  - APT 5 assigned
- Monday, April 9
  - Binary Trees
  - Lab 10 Due
  - Read 17.1 – 17.7