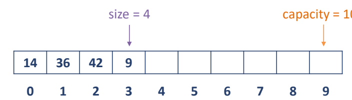
CSCI 262
Data Structures

15 – Linked Lists

CS@Mines

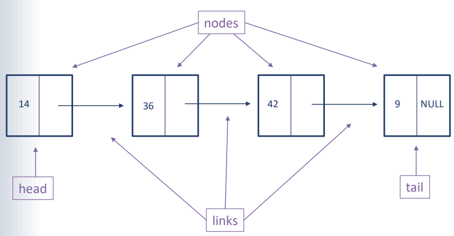
Array List (aka Vector)



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

CS@Mines

Linked Lists



Like the example array list on previous slide, this list contains {14,36,42,9}.

CS@Mines

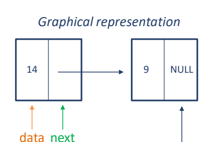
Node Class

Here's a very simple implementation of a node:

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

Pointer to node, inside node!

Graphical representation



Note NULL pointer in tail node!

CS@Mines

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

CS@Mines

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:

linked list passed as pointer to head node

tail->next is NULL, marks end of list

```
bool find(node* head, int val) {
    for (node* p = head; p != nullptr; p = p->next) {
        if (p->data == val) return true;
    }
    return false;
}
```

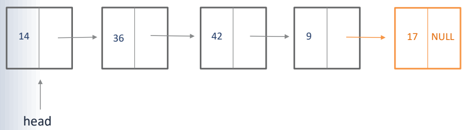
We only work with node pointers, thus the -> operator for accessing fields. Recall `p->data` is equivalent to `(*p).data`

CS@Mines

Linked List Operations: add

add:

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



CS@Mines

Linked List Operations: add

```
// 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;
```

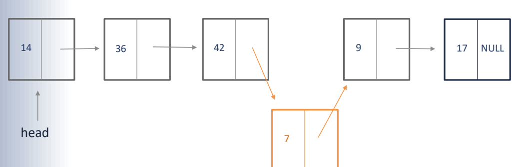
If we keep a *tail* pointer (in addition to head pointer), can skip first step; what is cost of add in each case?

CS@Mines

Linked List Operations: insert

insert:

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

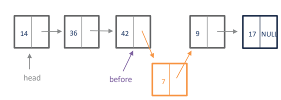


CS@Mines

Linked List Operations: insert

```
// 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?

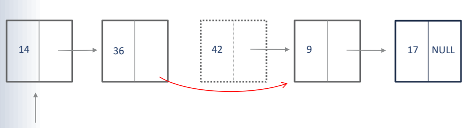


CS@Mines

Linked List Operations: erase

erase:

- Relink left node to right node
- Delete removed node



CS@Mines

Linked List Operations: erase

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

What is cost of erase?

CS@Mines

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*

CS@Mines

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?

CS@Mines

Efficiency: Array vs Linked

	Array	Linked
Add	O(1)	O(1)*
Insert	O(N)	O(1)†
Erase	O(N)	O(1)†
Indexed Get/Set	O(1)	O(N)
Append	O(N)	O(1)
…	Think about other operations you might use!	

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

CS@Mines

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():

```
void print_list(node* head) {
    for (node* p = head; p != NULL; p = p->next) {
        cout << p->data << endl;
    }
}

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

CS@Mines

Up Next

- Friday, November 2
  - Lab 10 – Queue, part 2
  - Project 3 due
  - APT 4 assigned
- Monday, November 5
  - Binary Trees
  - Lab 10 Due
  - Reading: Sections 16.1 – 16.2

CS@Mines