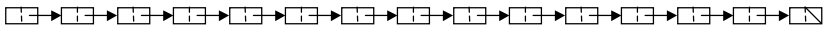
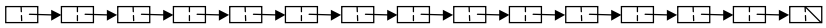


CSE 373: Stacks

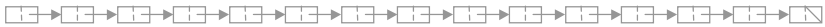
Chapter 3



Definition

Stack:

Stack Operations



Main Operations:

```
void push(Object);
Object pop();
Object top();
bool isEmpty();
```

Other Operations:

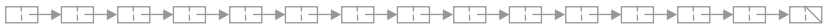
- normal creation/deletion operations
- generally no iteration operations (why?)

Stack Example

```
Stack<int> S;
int topval, newval;

S.push(1);
S.push(1);
for (i=2; i<n; i++) {
  topval = S.pop();
  newval = topval + S.top();
  S.push(topval);
  S.push(newval);
}
```

List-based Stack Implementation



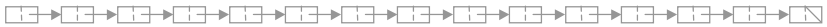
- Stacks are a specialized type of list
  - `push()` = `insert()` at a specific end of the list
  - `pop()` = `remove()` restricted to the same end
- Thus, Lists could be used to implement the Stack ADT
  - Advantages?
  - Disadvantages?

Array-based Stack Implementation

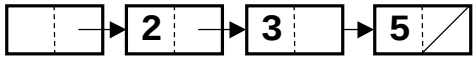
- *Recall:* what were the best/worst cases for `insert()/remove()` on array-based Lists?

- This implies a straightforward and efficient array implementation of Stacks
  - Advantages?
  - Disadvantages?

Link-based Stack Implementation



- *Recall:* `insert()` and `remove()` are cheap for link-based Lists once we locate the node that points to the new/old node



- What Link-based implementation of Stacks does this suggest?
  - Advantages?
  - Disadvantages?

Evaluating Stack Implementations

List-based *Array-based* *Link-Based*

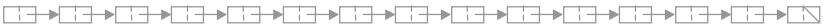
Operations:

```
push()
pop()
top()
isEmpty()
```

Space:

Other:

Applications of Stacks



- **compilers:** to represent scoped properties of languages

```
int a;
void (int x, int y) {
  int z;
  {
    int a,b;
    {
      int z;
    }
  }
```

- **graphics:** managing coordinate transformations (*e.g.*, OpenGL)
- **applications:**

(*hint:* you probably use one every time you use a Microsoft product)

Application: Function Call Stacks

```
void fact(int n) {
   … fact(n-1) …
}
void fowl(int z) {
  … cout << z …
}
void fish(int x,y) {
  … fowl(x) …
  … fact(y) …
}
void main() {
  … fish(3,5) …
}
```

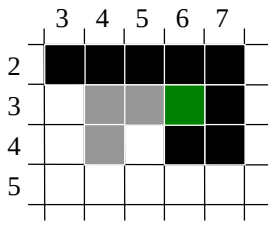
Application: Searching

Use a Stack to track where you've been:

e.g., `FillPaint()`:

- each element stores (x,y) & last direction we've tried
- assume we always search directions in a certain order
  - *e.g.*, N, E, S, W

(3,6) N
(3,5) E
(3,4) E
(4,4) N



Avoiding Calls to `new`

- Although `new` and `delete` have O(1) cost, the constant can be large enough that you want to avoid it
- One idea:
  - rather than calling `delete` on nodes, store them in a list
  - Then, before calling `new`, check to see if you can grab a node from the list instead