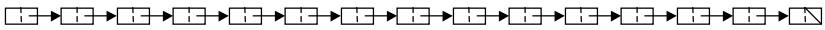
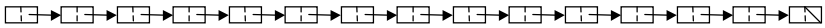


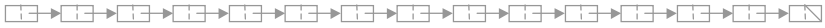
CSE 373: Stacks

Chapter 3

Stack:



Stack Operations



Main Operations:

```
void Push(Stack,SType);
SType Pop(Stack);
SType Top(Stack);
int IsEmpty(Stack);
```

Other Operations:

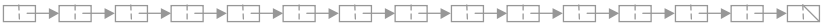
- normal creation/deletion operations
- generally no iteration operations (why?)

Stack Example

```
Stack S;
int topval, newval;

S = NewStack();
Push(S,1);
Push(S,1);
for (i=2;i<n;i++) {
  topval = Pop(S);
  newval = topval + Top(S);
  Push(S,topval);
  Push(S,newval);
}
```


List-based Stack Implementation



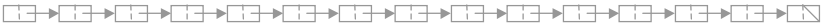
- Stacks are a specialized type of list
  - `Push()` = `Insert()` at a specific end of the list
  - `Pop()` = `Delete()` restricted to the same end
- Thus, Lists could be used to implement the Stack ADT
  - Advantages?
  - Disadvantages?

Array-based Stack Implementation

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

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

Link-based Stack Implementation



- *Recall:* `Insert()` and `Delete()` are cheap for link-based Lists once we locate the nodes that point to the node in question

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

Evaluating Stack Implementations

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

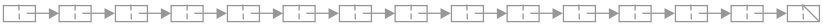
Operations:

```
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;
    }
  }
```

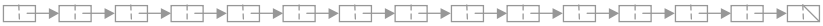
- **graphics:** managing coordinate transformations (*e.g.*, OpenGL)
- **applications:**

(*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) {
  … printf("%d",z);
}
void fish(int x,y) {
  … fowl(x) …
  … fact(y) …
}
void main() {
  … fish(3,5) …
}
```


Application: Searching

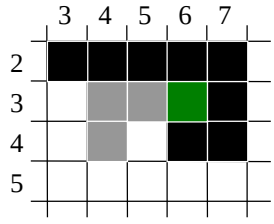


Use a Stack to track where you've been:

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

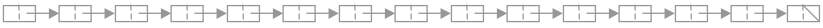


Avoiding Calls to `malloc()`

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

  (*This applies to* `new` *and* `delete` *as well…*)

Non-fixed size Array-based ADTs



`realloc()` can be used to resize a memory area returned by `malloc()`

- *possibly changing the pointer when doing so*
- copies values from the original area to the new one
- equivalent to doing a second `malloc()`, copying the data over, and calling `free()` on the original
- worst-case O(n) operation, due to the copy

`realloc()` is useful for creating an array-based List or Stack of arbitrary size