

## Lecture 7 CS107

exp stack of pointers to strings

```
int main ( --, --)
```

```
{ const char * friends[] = { "Abe", "Bob", "Carl" };
  stack_string Stack;
```

```
StackNew ( & string Stack, sizeof (char *));
```

```
for (int i=0; i<3; i++) {
  char * copy = strdup (friends[i]);
  StackPush ( &stringStack, &copy);
}
```

```
char * name;
```

```
for (int i=0; i<3; i++) {
  StackPop ( &stringStack, &name);
  printf ("%s\n", name);
  free (name);
}
```

```
StackDispose ( &stringStack)
```

suppose the preceding for loop was commented out ...

### Upgraded version

```
void StackNew (stack *s, int elemSize, void (*freefn) (void*))
```

```
void StackDispose (stack *s);
```

```
void StackPush (stack *s, void *elemAddr)
```

```
void StackPop (stack *s, void *elemAddr)
```

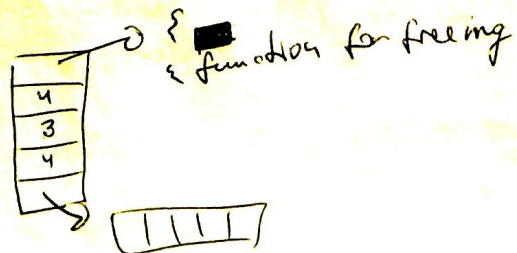
information about how to destroy elements when StackDispose is called  
NULL if a basic type.

stack.h

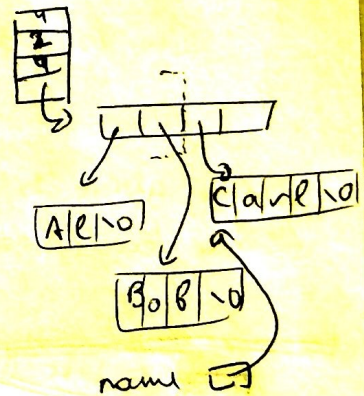
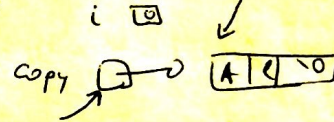
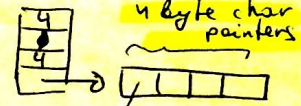
```
typedef struct {
  void * elems;
  int elemSize;
  int loglength;
  int allocLength;
  void (* freefn) (void *);
}
```

```
} stack
```

Store freeing function in struct



①



`void Stack Dispose (stack *s)`

Null if basic types are in the stack array

```
{
    if (s->freefn != NULL) {
        for (int i=0; i < s->length; i++) {
            s->freefn((char*)s->elems + i * s->elemSize);
        }
    }
    free(s->elems);
}
```

pointer to a function in the code evaluates function

`void String Free (void *elem)`

```
{
    free(*(char**) elem);
}
```

← strings case: dereference and free

Stack String Stack

Stack New (& string Stack, size of (char \*), String Free);

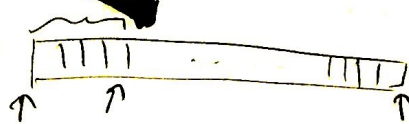
Rotate

`void rotate (void *front, void *middle, void *end)`

```
{
    int frontSize = (char*)middle - (char*)front;
    int backSize = (char*)end - (char*)middle;
    char buffer[frontSize];
    memcpy(buffer, front, frontSize);
    memmove(front, middle, backSize);
    memcpy((char*)end - frontSize, buffer, frontSize);
}
```

same trick to do arithmetic in terms of bytes

buffer with exact # of bytes



← cannot use memcpy because overlapping and memcpy does not account for it

could memmove only if have to because the source and destination regions could overlap



## Lecture 7 CS107

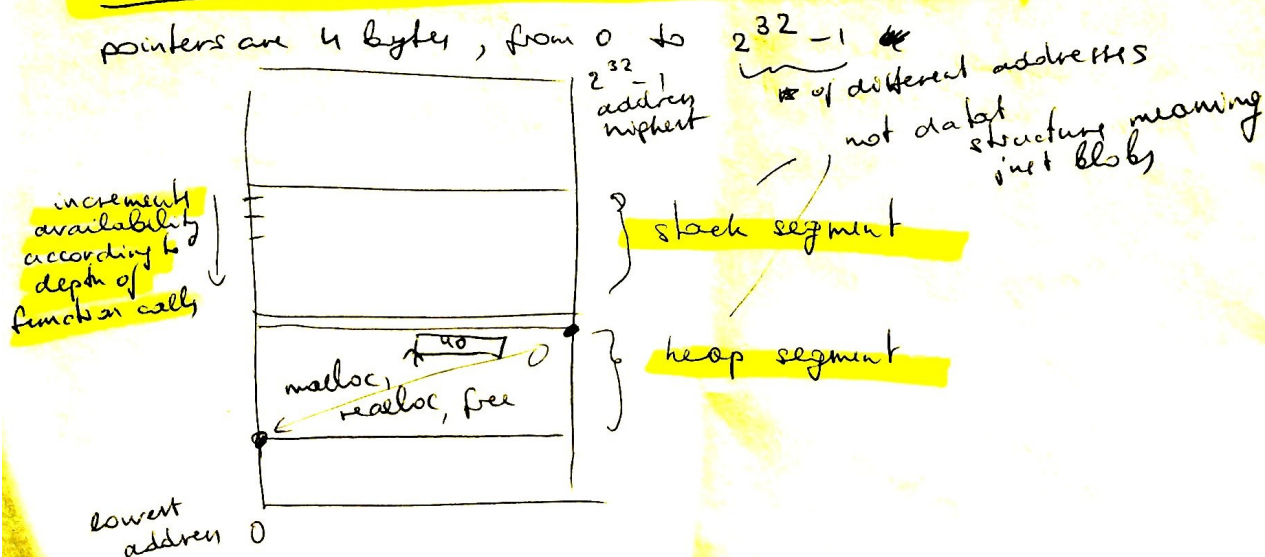
2

### Quicksort

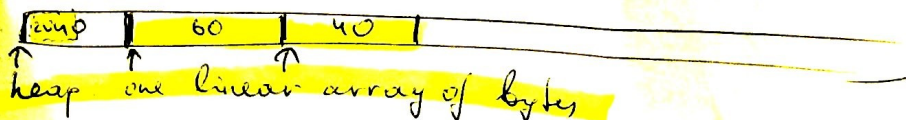
```
void qsort (void *base, int size, int elemSize,  
            int (*cmpfn) (void *, void *));
```

=  
> non qsort  
> non bsearch  
memcpy  
memmove  
malloc  
realloc  
free

### Implementation of malloc, realloc and free



### Focus on heap



`void *a = malloc(40);` 1) starts at the beginning of the heap and searches for an array segment of 40 bytes requested size  
`void *b = malloc(60);` 2) records that the segment is in use  
`free(a);` records that the \*a blob is available, leaves bit pattern

void \*c = malloc(44); will look at the first  
void \*d = malloc(20); 40 block but the proceed

Data Structure used by heap manager

link list of free nodes



Size of node  
point to next

Traverse the linked list to  
determine which node can accommodate  
a malloc request