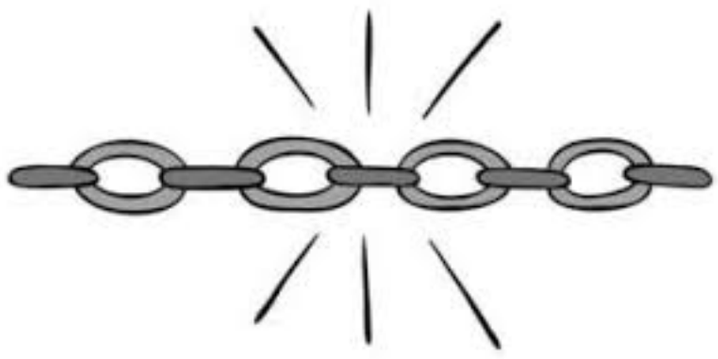
Lecture 13
Data Structures



Problem

Suppose we have a long list of integers we would like to examine and manipulate

position	*list items*
1	3
2	14
3	7
4	8
5	23
6	5
.	.
.	.
.	.

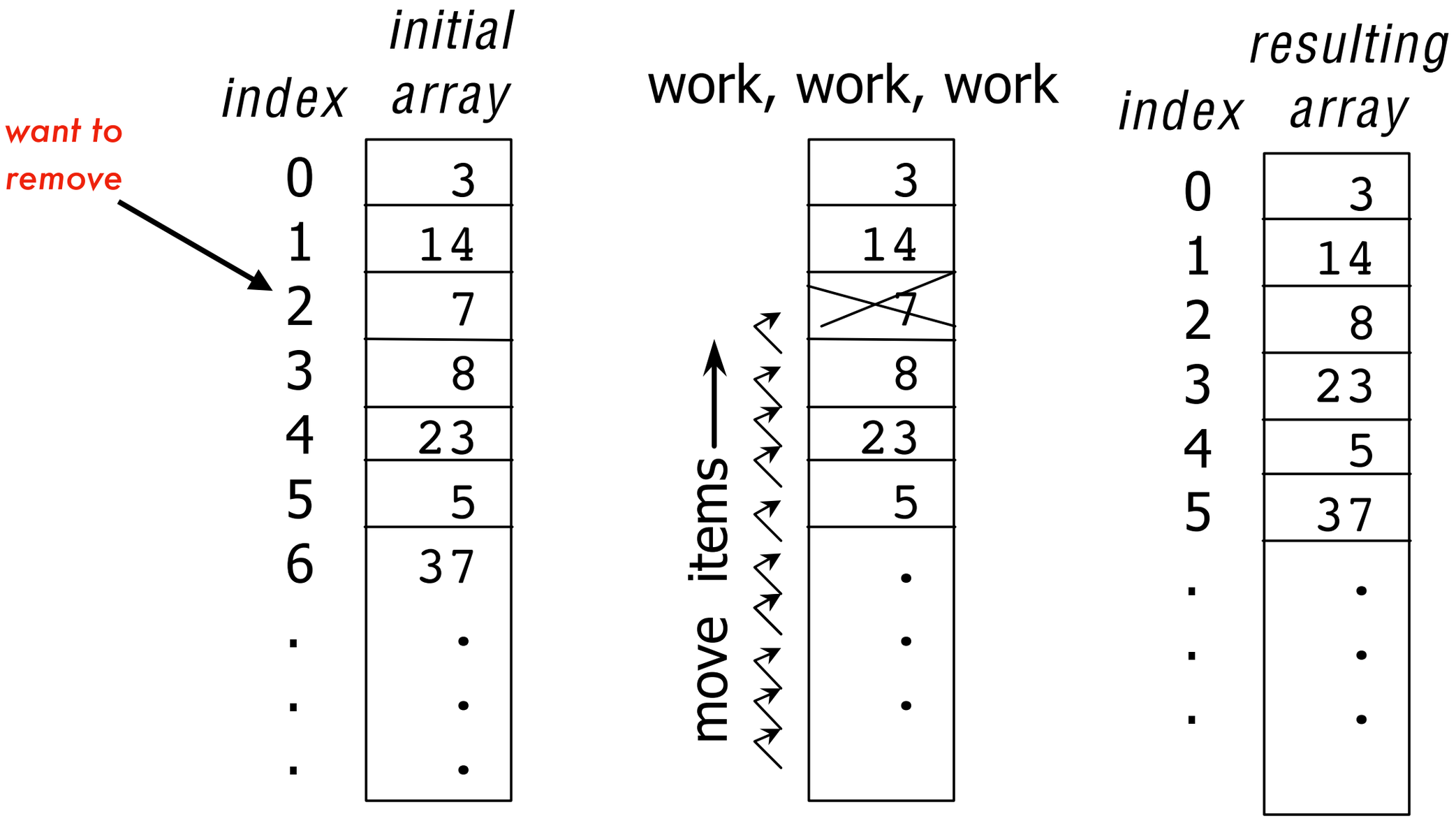
Might choose to represent this data using an array

Each element occupies 4 bytes in memory

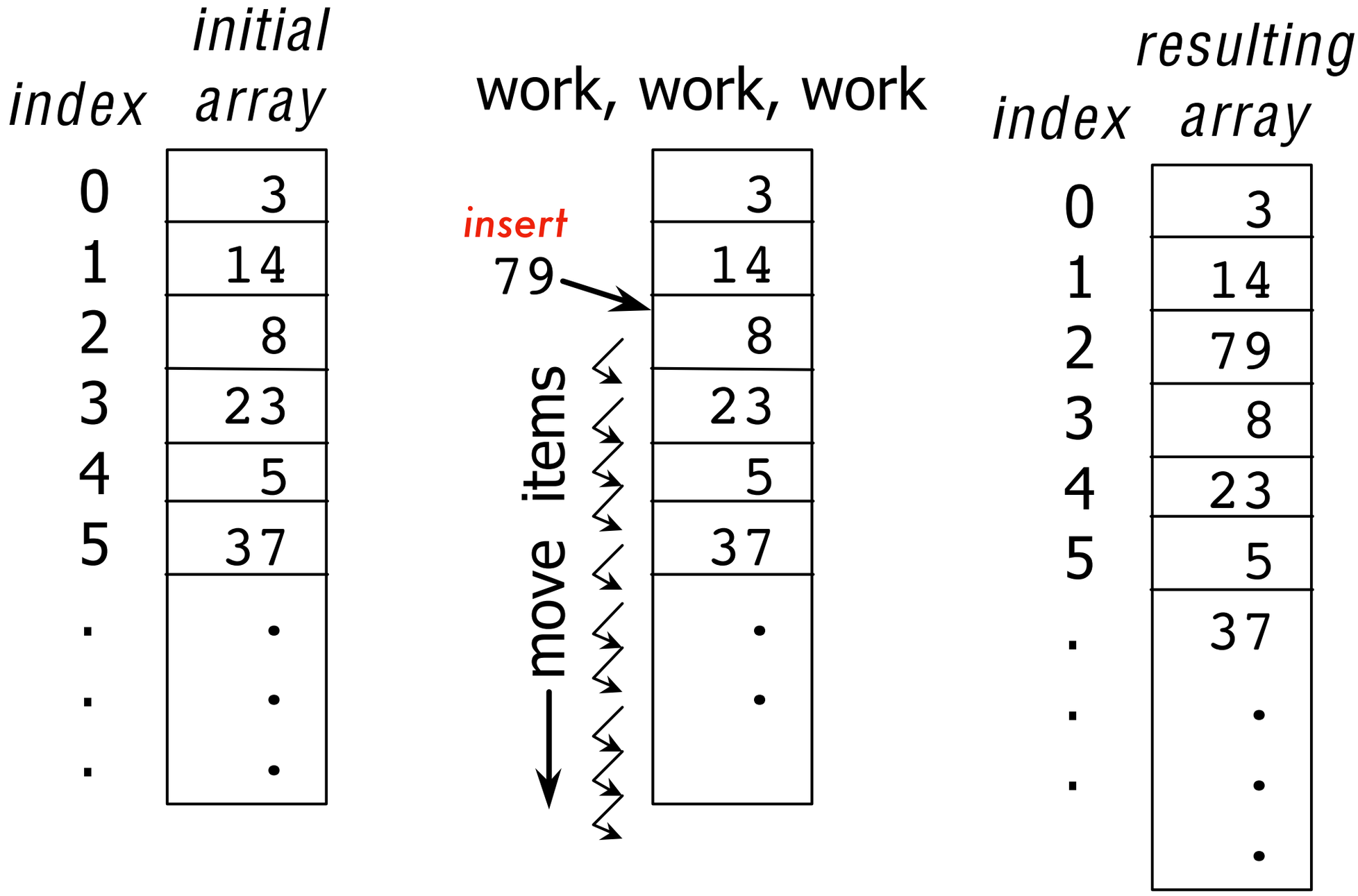
index	*array elements*
0	3
1	14
2	7
3	8
4	23
5	5
.	.
.	.
.	.

Issues

- Arrays are not a useful data structure if
  - the number of elements is expected to change
  - the "shape" of the structure is expected to change



Issues


initial
index
array
0
3
1
14
2
8
3
23
4
5
5
37
work, work, work
insert
79
move items
resulting
index
array
0
3
1
14
2
79
3
8
4
23
5
5
37

Issues

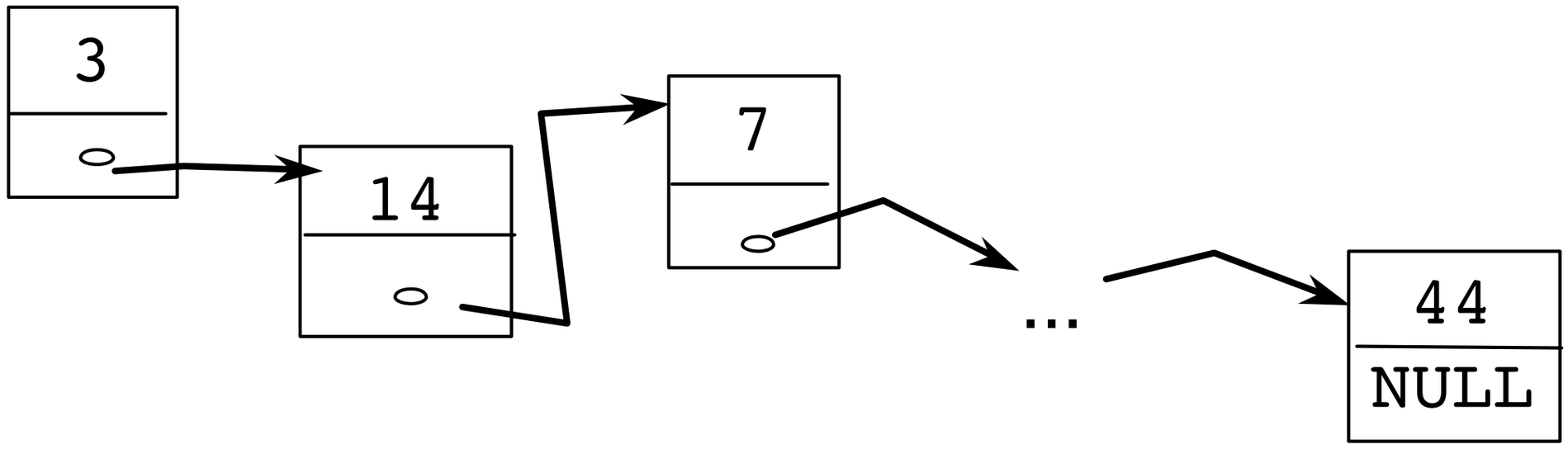
- What about performance and efficiency?
  - Irregular population of the array …
  - Predetermination of maximum number of elements
    - Maybe difficult to assess at compile-time

A linked-list is a data structure whose elements are allocated dynamically, thereby alleviating these drawbacks

Linked-List

A sequence of nodes in which:

- Each node contains a data element
- Every node but the last contains a pointer to another node
- The first node in the sequence is not pointed-to by any other node
- The last node in the sequence has its "next-node" field set to NULL



- Notably, there is no requirement that nodes are contiguous in memory
- What are its drawbacks compared to using arrays?

Linked-List

- Represent the "next-node" field as a pointer
- Allocate new nodes using `malloc()`
- Remove nodes using `free()`

```
struct   lnode
{
    int         item;
    struct      lnode *next;
};
```

This describes the components of the `struct`;
no memory is allocated

Now we could write

```
struct   lnode *np;
np = (struct lnode*)malloc(sizeof(struct lnode));
np->Item = 3;
np->Next = NULL;
free(np);
```

allocate memory for a pointer

allocate memory for a node

deallocate node's memory

Operations

- Represent a node using a struct
- Insert a new node into a list
  - involves node creation using malloc()
- Remove a node from the list
  - involves deallocation using free()

Dynamic creation

```
struct l node *np;

np =(struct lnode*)malloc(sizeof(struct lnode)) ;
```

Dynamic deallocation

```
free(np);
```

Type Definition


```
typedef struct Node {
    void *data;
    struct Node *next;
} Node;

// Linked list structure
typedef struct {
    Node *head;
    size_t element_size; // Size of each element in the list
} LinkedList;

// Function to initialize a new linked list
void initLinkedList(LinkedList *list, size_t element_size) {
    list->head = NULL;
    list->element_size = element_size;
}
```

Insert


```
void insertBeforeNode(LinkedList *list, Node *nextNode, void *data) {
    Node *newNode = (Node *)malloc(sizeof(Node)); // new node
    newNode->data = malloc(list->element_size); // space for data

    // copy contents of *data to newNode->data; why?
    for (size_t i = 0; i < list->element_size; i++) {
        *(char *)(newNode->data + i) = *(char *)(data + i);
    }


    if (nextNode == list->head) {  // insert at the beginning
        newNode->next = list->head;
        list->head = newNode;
    } else {  // search for node
        Node *temp = list->head;
        while (temp != NULL && temp->next != nextNode) {
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Error: Node not found in the list.\n");
            return;
        }
        newNode->next = temp->next;  // temp points to correct node
        temp->next = newNode;  // insert newNode into the list
    }
}
```

Delete


```
void freeNode(LinkedList *list, Node *nodeToRemove) {
    if (nodeToRemove == NULL) {
        return;
    }

    Node *temp = list->head;

    // If the node to be removed is the head node
    if (temp == nodeToRemove) {
        list->head = temp->next;
        free(temp->data);
        free(temp);
        return;
    }

    // Traverse the list to find the previous node
    while (temp != NULL && temp->next != nodeToRemove) {
        temp = temp->next;
    }

    // If the node was not found
    if (temp == NULL) {
        printf("Error: Node not found in the list.\n");
        return;
    }

    // Update the next pointer of the previous node to skip the current node
    temp->next = nodeToRemove->next;
    free(nodeToRemove->data);
    free(nodeToRemove);
}
```

Other variations

- Traverse the list from tail to head
- Exchange elements in constant-time
- Insert an element before another in constant-time

A doubly-linked list extends a singly-linked list with both a "previous-node" as well as a "next-node" pointer

```
struct double_linked_list {
  int value;
  struct double_linked_list *next_ptr;
  struct double_linked_list *prev_ptr;
 };
```

Stack


```
typedef struct node {
  struct node *next;
  void *data;
} node;
```

Each stack element stores a pointer to the next node in the stack and a pointer to the data the node stores.

A stack contains meta-data on the number of elements, the size of each element, and a pointer to the current 'top-of-stack'

- A stack is generic, but not heterogeneous

```
typedef struct stack {
  int nelems;
  int elem_size_bytes;
  node *top;
} stack;
```

Operations:

- `stack *stack_create(int elem_size_bytes)`
- `void stack_push(stack *s, const void *data)`
- `void stack_pop(stack *s, void *addr)`

Stack create


```
stack *stack_create(size_t elem_size_bytes) {
  stack *s = malloc(sizeof(stack));
  s->nelems = 0;
  s->top = NULL;
  s->elem_size_bytes = elem_size_bytes;
  return s; }
```

Stack


```
void stack_push(stack *s, const void *data) {
  // allocate space for the new node
  node *new_node = malloc(sizeof(node));
  // allocate space for the node's data
  new_node->data = malloc(s->elem_size_bytes);
  // copy data from argument; why?
  memcpy(new_node->data, data, s->elem_size_bytes);
  new_node->next = s->top;
  s->top = new_node;
  s->nelems++;
}
```

Stack


```
void stack_pop(stack *s, void *addr) {
  if (s->nelems == 0) {
        // panic: trying to pop an empty stack   }
    node *n = s->top;
    // copy contents of node to *addr
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;
    // free data associated with this node
    free(n->data);
    free(n);
    s->nelems--;
}
```

Stack Usage


```
stack *intstack = stack_create(sizeof(int));

int num = 7;
stack_push(intstack, &num);
```

supply address containing data to be pushed

```
int popped_int;
while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
printf("%d\n", popped_int); }
```

supply address where popped data should be copied

A More Efficient Stack


```
typedef struct stack {
    size_t nelems;
    size_t elem_size_bytes;
    void *top; } stack;
```

- Avoid allocating a struct for stack nodes
- `Inline` data directly into the stack structure
- A node is just a contiguous set of memory bytes of memory storing (1) address of next node, and (2) data

Stack create


```
stack *stack_create(size_t elem_size_bytes) {
  stack *s = malloc(sizeof(stack));
  s->nelems = 0;
  s->top = NULL;
  s->elem_size_bytes = elem_size_bytes;
 return s; }
```

Since no node is involved in creating a stack, nothing changes from the original definition

Stack push


```
void stack_push(stack *s, const void *data) {
// A node contains a pointer to the next node on  the stack and allocated space for data
void *newNode = malloc(sizeof(void *) + s->elem_size_bytes);
    // Copy data into node - 'upper' part contains data, 'lower' part contains pointer to next node
    memcpy((char *) newNode + sizeof(void *),
           data, s->elem_size_bytes);
    //  The contents at the address pointed to by newNode is a pointer pointing to the current top
    *((void **) newNode) = s->top;
    s->top = newNode;
    s->nelems++;
}
```

Stack pop


```
void stack_pop(stack *s, void *addr) {
  if (s->nelems == 0) {
     //panic: trying to pop an empty stack
    }
   void *n = s->top;
   memcpy(addr, (char *) n + sizeof(void *), s->elem_size_bytes);
s->top = *(void **) n;
free(n);
s->nelems--;
```