

Recap

- Last time we discussed:
 - Automatic vs. dynamic memory allocation
 - malloc family of functions
 - calloc
 - realloc

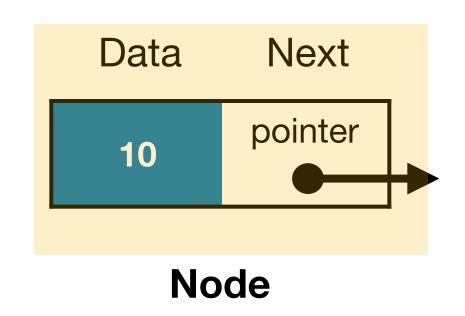
- Calling free to release memory
- Allocating 2D arrays
- Memory leak vs. seg-faults
- valgrind to detect memory leaks.

Today - linked list

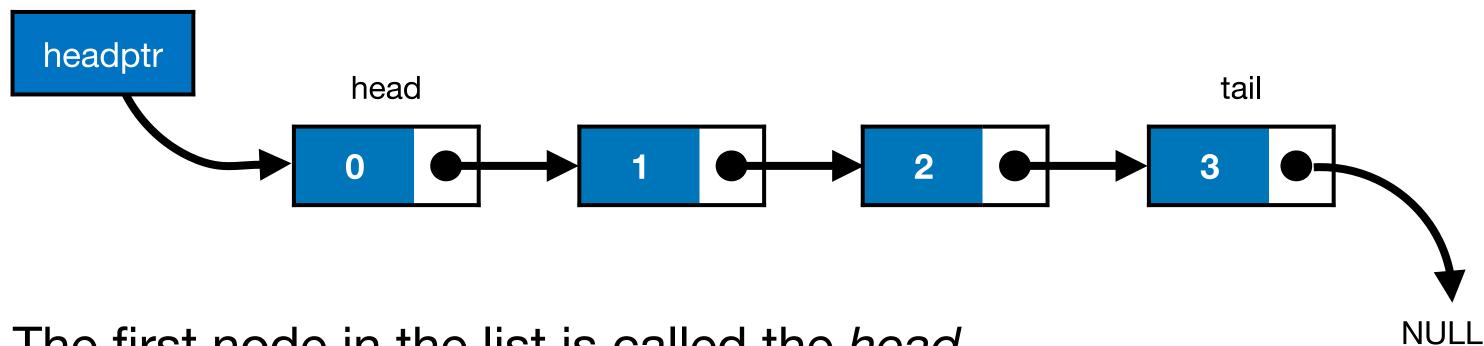
- What is a list ... really?
 - A list is collection of elements/items which can be accessed sequentially.
 - Entertains the concept of order; first, second, last.
 - Note: An empty list is still a list.
- An array is an indexed list; i.e. can access elements by their index.

Linked list

- A <u>linked list</u> is an *ordered* collection of items (often called *nodes*), each of which contains some data, connected using *pointers* (hence the link part).
- A node is a collection of two sub-elements or parts.
 - A data part that stores the actual element
 - And a *next* part (pointer) that stores the address of the next node.

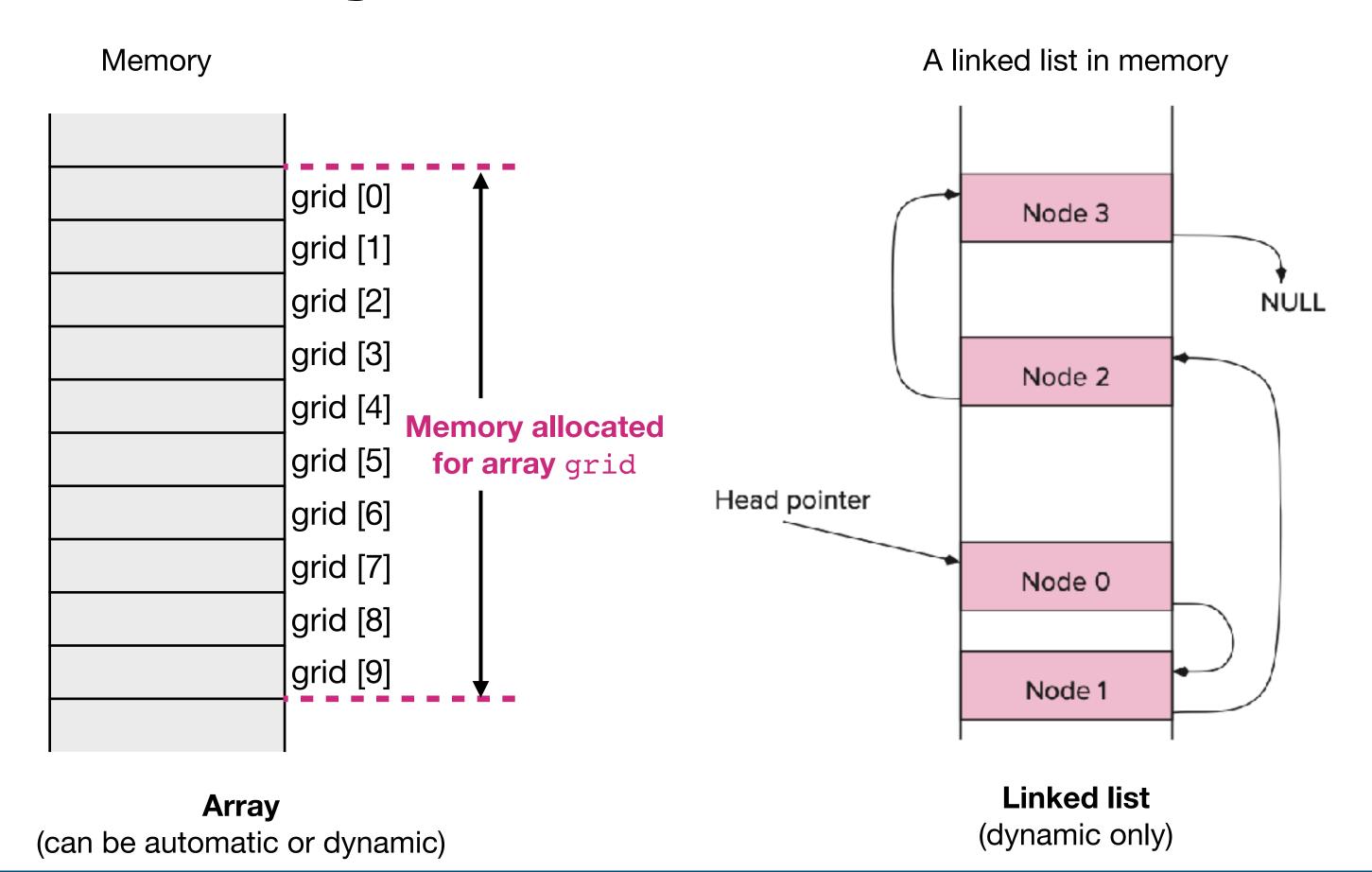


Linked list



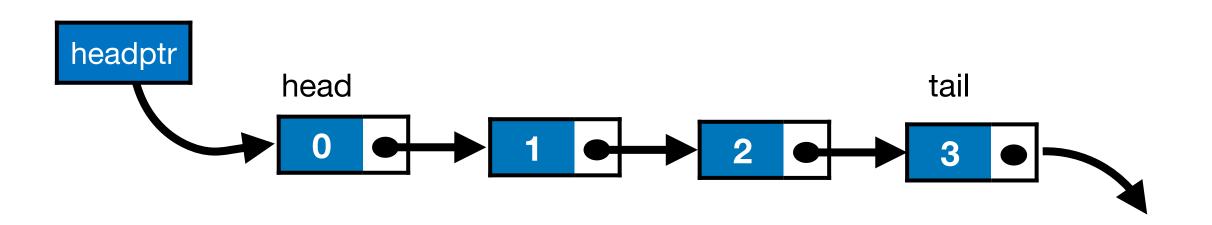
- The first node in the list is called the head
 - Accessed using a pointer called head pointer
 - Used as the starting reference to traverse the list
- The last node in the list is called the tail.
 - The tail may contain data, but it always points to NULL value

Array vs. linked list



Array vs. linked list

Element 0
Element 1
Element 2



	Array	Linked list
Memory Allocation	Automatic / Dynamic	Dynamic
Memory Structure	Contiguous	Not necessarily consecutive
Order of Access	Random	Sequential
Insertion / Deletion	Create/delete space, then shift all successive elements	Change pointer address

NULL

Basic operations

- Inserting an item in the list
 - Unsorted list: Can insert at <u>head</u> or at <u>tail</u>
 - Sorted list: Insert so as to <u>maintain</u> sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from <u>head</u>, <u>tail</u> or <u>by key</u>.

Example: Student record

```
typedef struct StudentStruct{
   int UIN;
   char *netid;
   float GPA;
}student;
```

Using structs

```
typedef struct StudentStruct{
   int UIN;
   char *netid;
   float GPA;
   struct StudentStruct *next;
}node;
```

Using linked lists

Example: A person

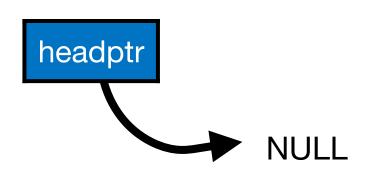
```
typedef struct person{
    char *name;
    unsigned int birthyear;
}Person;
```

Using structs

```
typedef struct person{
    char *name;
    unsigned int byear;
    struct person *next;
}node;
```

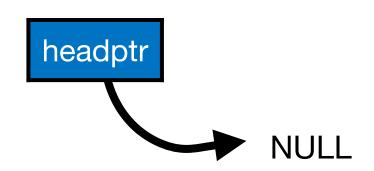
Using linked lists

 What should be the empty list?

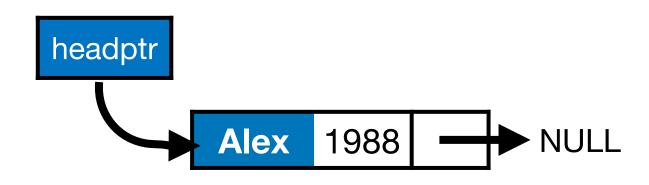


```
typedef struct person{
    char *name;
    unsigned int byear;
    struct person *next;
}node;
```

 What should be the empty list?



What should be the singleton list?



```
typedef struct person{
    char *name;
    unsigned int byear;
    struct person *next;
}node;

node* headptr;
node* temp=(node*) malloc(sizeof(node));
temp->name="Alex"
temp->byear=1988;
temp->next=NULL;
headptr = temp;
```

Linked lists - more elements



Suppose we want to add another node

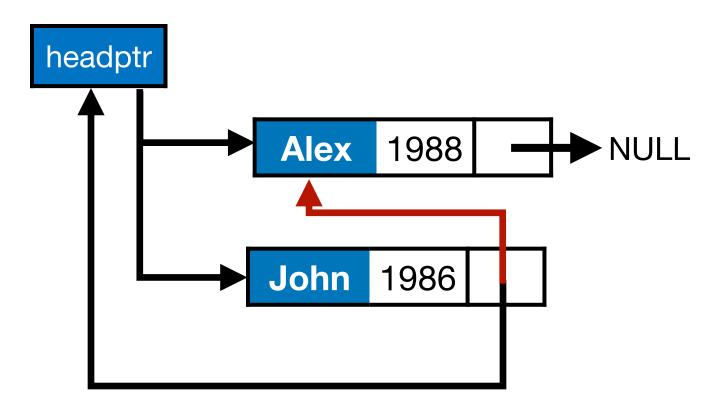
```
{"John", 1986, }
John 1986
```

- Should the node be added at the head or tail?
 - For sorted linked lists, this node should go at the head
 - For plain linked lists, we get to choose.

- Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from head, tail or middle.

Linked lists - adding a node

- Suppose we want to add at head.
- What needs to be done?
 - New node should point to current head.
 - Current head should be updated to new node.



- Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from head, tail or middle.

Linked lists - adding a node

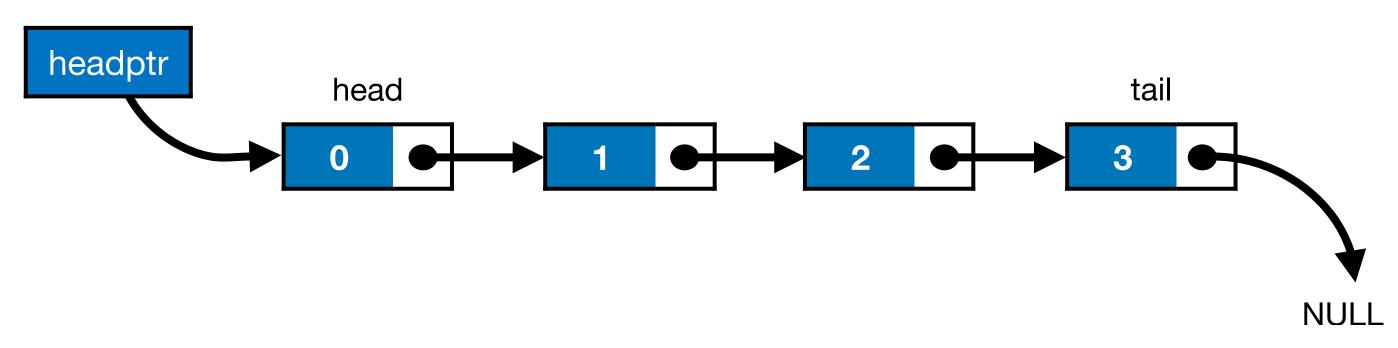
- Suppose we want to add at head.
- What needs to be done?
 - New node should point to current head.
 - Current head should be updated to new node.
 - Deal with case of empty list

```
In our code, cursor will stand for the node currently being examined; in this example the head pointer

temp->next = cursor;
cursor = temp;
```

node* temp=(node*) malloc(sizeof(node));

Traversing a linked list



- Head pointer points to the first node of the list.
- To traverse the list we do the following
 - Follow the pointers.
 - Display the contents of the nodes as they are traversed.
 - Stop when the next pointer points to NULL.

Linked lists - traversing

- Recall that linked lists are defined recursively. So to traverse and print.
 - If the list is empty do nothing,
 - otherwise, print current element &
 - recurse on the rest!

- Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from head, tail or middle.



Exercise

- Let us put together whatever we tried so far.
- Add the following nodes successively to the head of an empty list and print the list out.
 - {Alex, 1988}
 - {John, 1986}

- {Mary, 1990}
- {Sue, 1992}

• Functions to write (a) print_list to traverse node and (b)
add_at_head to add to head.

Code so far ...

```
void add_at_head(node *cursor, node *new){
  node *temp = malloc(sizeof(node));
  temp->name = new->name;
  temp->byear = new->byear;
  if (cursor == NULL)
    cursor = temp;
  else{
    temp->next = cursor;
    cursor = temp;
```

What happened?

What happened?

```
void add_at_head(node **cursor, node *new)
```

```
node * temp = (node *) malloc(sizeof(node));
temp->name = new->name;
temp->next = new->next;
```

headptr is a single pointer that should always point to start of list. Since we are relying on a function to make an update, we need to pass-by-reference (remember the defective swap function?)

```
if (*cursor == NULL)
  *cursor = temp;
else{
  temp->next = *cursor;
  *cursor = temp;
}
```

Since we are passing in a double pointer the code from slide #20 had to be carefully updated to make the types match as done above.

An pointer to new is passed to add_at_head. We copy that onto the heap so that the calling function can/may reuse the parameter it passed in.

```
if (cursor == NULL)
    cursor = temp;
else{
    temp->next = cursor;
    cursor = temp;
}
```

Adding a node - add at tail

- A pure implementation of a singly linked-list is completely defined by its head pointer.
 - Aside: A doubly linked lists has a pointer to the next element as well as the previous element (... tune in next week)
- To add an item at the tail position, we need to first find the tail.
 How: The only element in the list whose next is NULL is the tail element.
 - Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
 - Traversing the list
 - Deleting an item from the list
 - Delete from head, tail or middle.



Adding at tail

Just like
 print_list, keep
 traversing/recursing
 till tail element is
 found. Then add the
 new node there.

```
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

Note: We don't keep adding large blocks on the stack in this version because we are passing around a *pointer* to new. **This is important!**

If we did not do that, then recursion could overflow available space on the stack very quickly!

Deleting a node from head

- To delete a node from the head is simple.
 - Make a copy of the head pointer
 - Shift the head pointer to its next item
 - Call free on a copy of the head pointer
- What if list empty?

Exercise: Can we delete the entire linked list with just this function?

```
node *old_head = *headptr;
*headptr = (*headptr)->next;
free(old_head);
```

- Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from **head**, tail or middle.



Deleting the tail node

- To delete a node from the tail is more involved.
 - First find the second to last node - how?
 - Call free on second_last elements next.
 - Set second_last's next to NULL.
 - What if list empty?
 - What if singleton list?

```
void del_tail(node **cursor){
```

```
node * second_last = *cursor;
while (second_last->next->next != NULL)
   second_last=second_last->next;
free(second_last->next);
second_last->next = NULL;
```

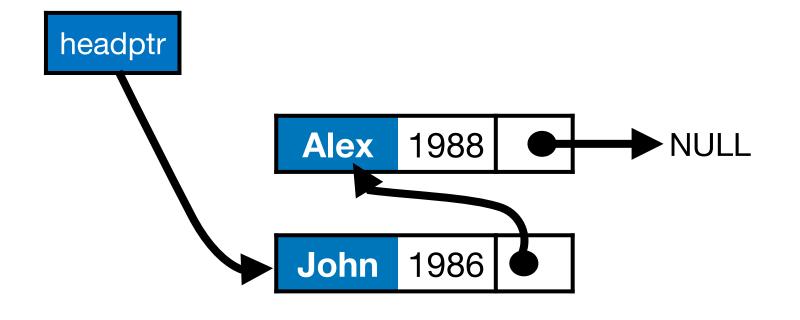
- Traversing the list
- Deleting an item from the list
 - Delete from head, tail or middle.

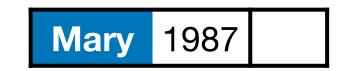


 Suppose our linked list is already sorted by birth year.

Give a new node, how to find its insertion point?

Let us start from basics!



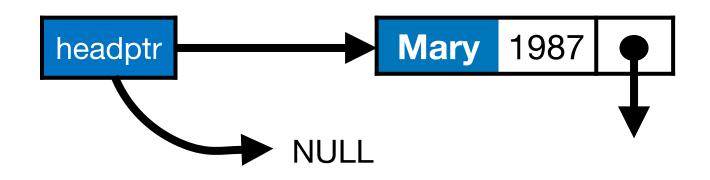


- Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from head, tail or middle.

 Suppose our linked list is already sorted by birth year.

void insert(node **cursor, node *new){

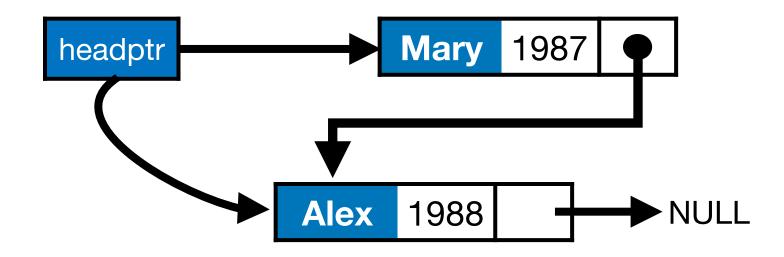
Give a new node, how to find the its insertion point?



If empty list, add at head.

 Suppose our linked list is already sorted by birth year.

Give a new node, how to find the its insertion point?



What if not empty?

```
void insert(node **cursor, node *new){
  if ((*cursor == NULL) ||

    add_at_head(cursor, new);
    return;
}
```

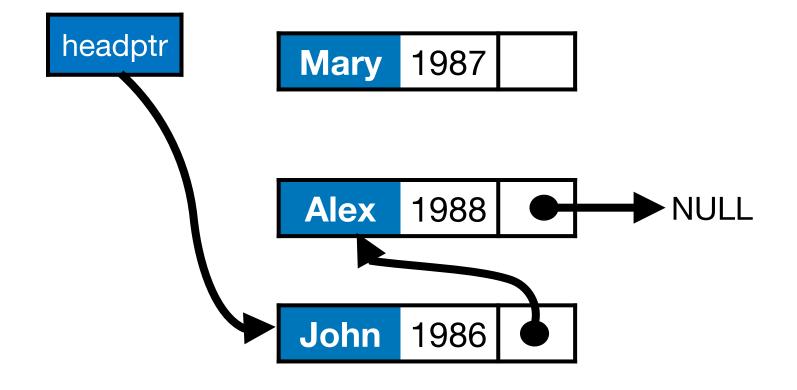
If first item is bigger than new node still add at head!



 Suppose our linked list is already sorted by birth year.

Give a new node, how to find the its insertion point?

```
void insert(node **cursor, node *new){
  if ((*cursor == NULL) ||
      (*cursor)->byear>=new->byear){
    add_at_head(cursor, new);
    return;
}
```



General case: if list is not empty and first item is smaller than new, update pointer & recurse!

- Inserting an item in the list
 - Unsorted list: Can insert at head or at tail
 - Sorted list: Insert so as to maintain sorted property
- Traversing the list
- Deleting an item from the list
 - Delete from head, tail or middle.

Deletion

- To delete a node we have to specify it by some identifying quantity.
- Then we traverse/search through the list. Cases are:
 - Item not found
 - Item found at head
 - Item found elsewhere



Search

- Left as an exercise ... should be easy enough now that you have seen how to look for, find and then delete a node!
 - Note: When an element is found, there is no index to return; so what should the search function do?
 - What to return when element is not found in list?

Dr. Ivan Abraham