ECE264: Advanced C Programming

Summer 2019

Week 5: Examples of Recursive Algorithms (Mergesort, Depth-first search, Enums, Unions, Complex Structures, Dynamic data structures (Linked lists, Stacks, Queues)

Merge sort

- Based on the principle of divide-conquer
  1. Divide the array into roughly two equal halves (sub-arrays)
  2. Sort the divided sub-arrays separately
  3. Merge the sorted sub-arrays to produce the larger array
- Always takes the same amount of time to run (best-case or worst-case)

Recursive code skeleton

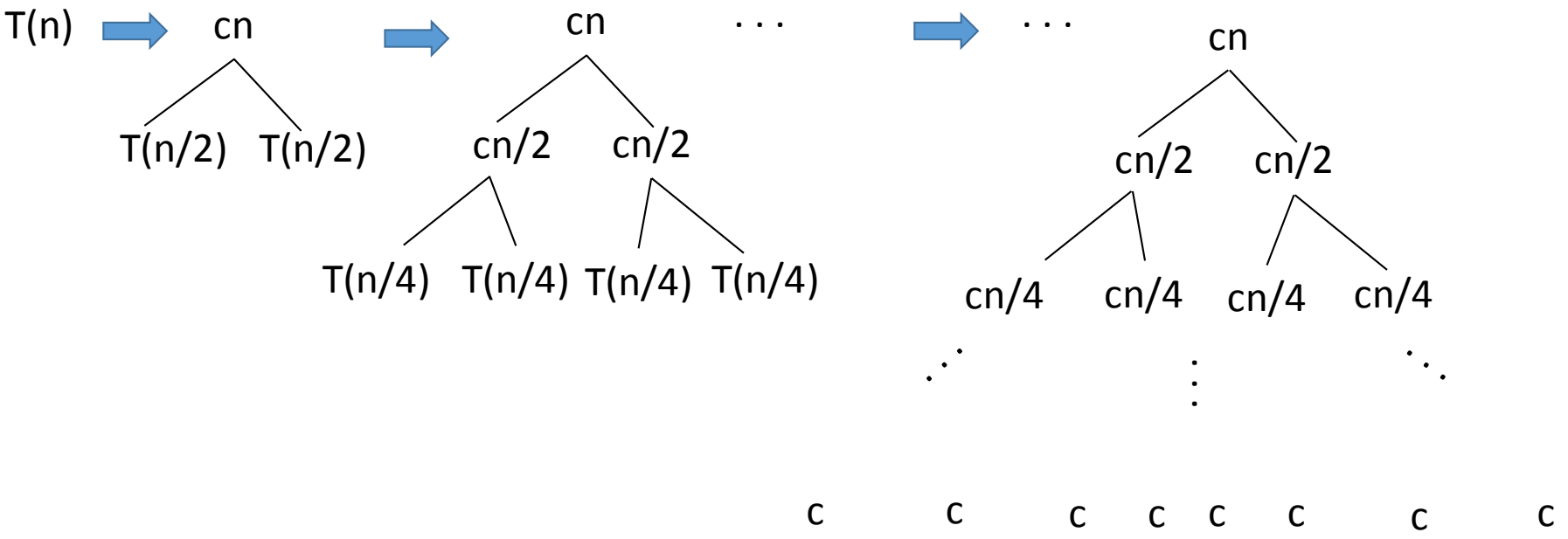
```
void Mergesort(int* arr, int left, int right) {
      if (left >= right)
            return;
      //compute middle index. Left-subarray
always >= right sub-array
      int nels = (right – left + 1) / 2;
      int nelsLeft = (nels + 1) / 2 ;
      int mid = left + nelsLeft - 1;
     //Recursively sort
      Mergesort(arr, left, mid)
      Mergesort(arr, mid+1, right);
      Merge(arr, left, mid, right);
}
```

Recursive code skeleton

```
void Merge(int* arr, int l, int m, int r) {
1 numL = (m – l + 1); numR = (r – m);
2 //reserve space for 1 element more than numL and numR
  in left- and right-subarrays L and R
4 //copy arr[l] to arr[m] into L[0] to L[numL-1]
5 //copy arr[m+1] to arr[r] into R[0] to R[numR-1]
6 i=0;j=0; //initialize indices to L and R
7 L[numL]=INFINITY; R[numR]=INFINITY;
8 for(p=l; p<=r; p++) {
9   if(L[i] <= R[j]) {
10    arr[p]=L[i]; i++;
11    } else {
12    arr[p]=R[j]; j++;
13    }
14   }
15 }
```

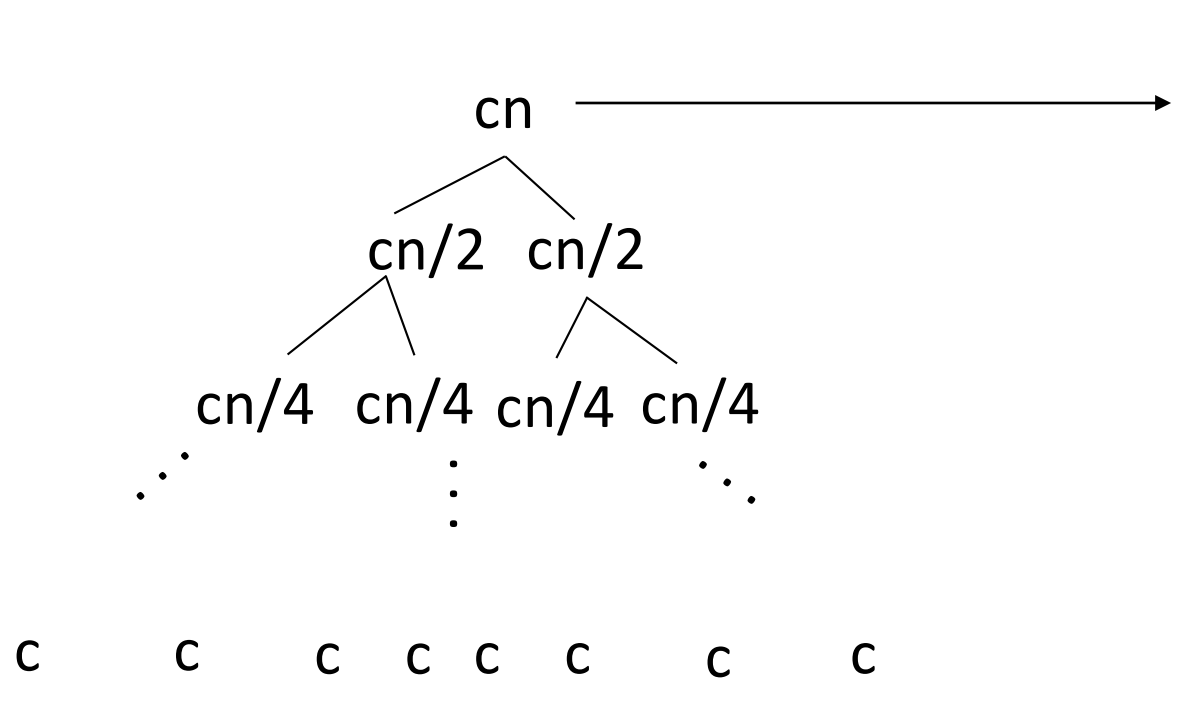
Merge sort analysis

- Assume n is perfect power of two



Merge sort analysis

- Assume n is perfect power of two



Level	Cost
0	cn
1	cn
2	cn
.	.
.	.
$\log_2 N$	cn

$= cn(\log_2 n + 1)$

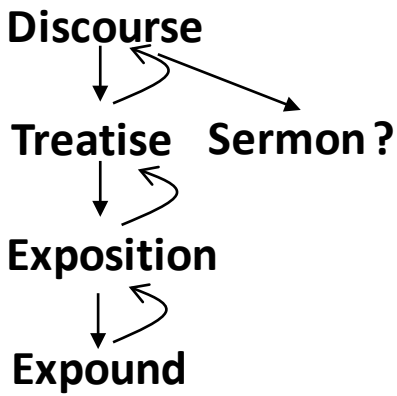
$= cn\log_2 n + cn$

Recursion – more examples

- Depth first search
  recall dictionary lookup:

"This is an increasingly common occurrence in our political **discourse**."
Washington Post Jun 25, 2019

discourse: a formal discussion of a subject in speech or writing, as a dissertation, **treatise**, sermon, etc.
treatise: a formal and systematic **exposition** in writing of the principles of a subject, generally longer and more detailed than an essay.
exposition: the act of **expounding**, setting forth, or explaining.
expound: To set forth or state in detail



Depth first search

We looked up the *first word we did not know* until we knew a definition completely. Then, we started backing up.

- Recursive procedure!
- One more example in PA?? (next week)

Enumerations (enums)

- Way to create user defined data type
- An alternative to using *magic numbers*
  - *Gives names to numbers – makes it easier to read and maintain programs*

```
typedef enum {<const1>, <const2>, … <const n>}
<typename>;
```

Enumerations (enums)

```
enum days{MON,TUE,WED,THU,FRI,SAT,SUN};
```

1. This **defines** a type called days

```
typedef enum {MON,TUE,WED,THU,FRI,SAT,SUN} days;
```

2. This **defines** a type called days (same as 1)

```
days dayOfWeek;
```

- This **defines** a variable called dayOfWeek, whose type is days

Enum - example

```
#include<stdio.h>
typedef enum days{MON,TUE,WED,THU,FRI,SAT,SUN};
int main(){
  days day = MON;
  days later = WED;
  days nineDaysLater = ?; //insert code here.
  printf("Now is %d",day);//prints "Now is 0"
  printf("Later is %d",later);//prints "Later is 2"
}
```

Enum – contd..

```
typedef enum days{MON,TUE,WED,THU,FRI,SAT,SUN};
```

Value = 0, 1, 2, 3, 4, 5, 6

```
typedef enum days{MON=100,TUE,WED,THU,FRI,SAT,SUN};
```

Value = 100, 101, 102, 103, 104, 105, 106

```
typedef enum days{MON,TUE,WED,THU=100,FRI,SAT,SUN};
```

Value = 0, 1, 2, 100, 101, 102, 103

Unions

- Another way to create user defined data type

```
typedef union {
    int i;
    double d;
    char c;
}mpm;
```

Syntax is very similar to structure/enum definition

Union – accessing members

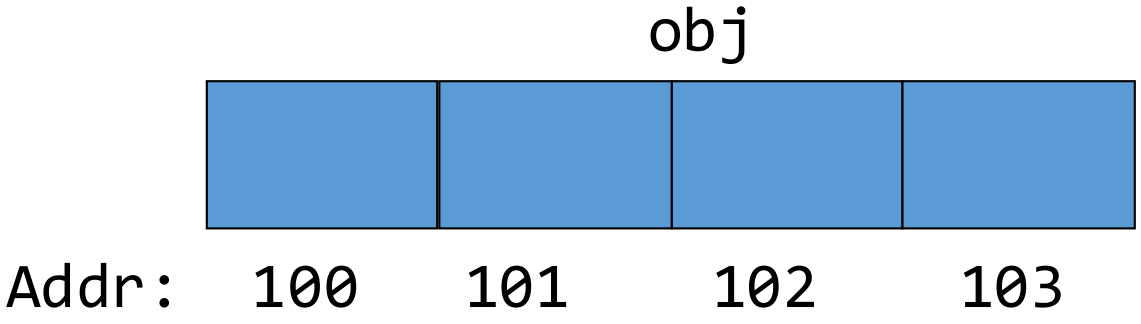
```
#include<stdio.h>
typedef union{
  int i;
  double d;
  char c;
}mpm;
int main(){
   mpm unionVar;//declaring an object of type mpm
   unionVar.i = 10;//accessing members of unionVar
   printf("%d",unionVar.i);//prints "10"
   unionVar.d = 3.14;
   printf("%f",unionVar.d);//prints "3.14"
   unionVar.c = 'A';
   printf("%c",unionVar.c);//prints A
}
```

Union – overwriting memory

```
int main(){
   mpm unionVar;
   unionVar.i = 0x12345678;
   printf(“%x”,unionVar.i);//prints “12345678”
   unionVar.d = 3.14;
   printf(“%f”,unionVar.d);//prints “3.14”
   unionVar.c = ‘A’;
   printf(“%c”,unionVar.c);//prints A
   printf(“%x”,unionVar.i);//prints 51eb8541
   printf(“sizeof(mpm):%zu\n”,sizeof(mpm));
//prints 8
}
```

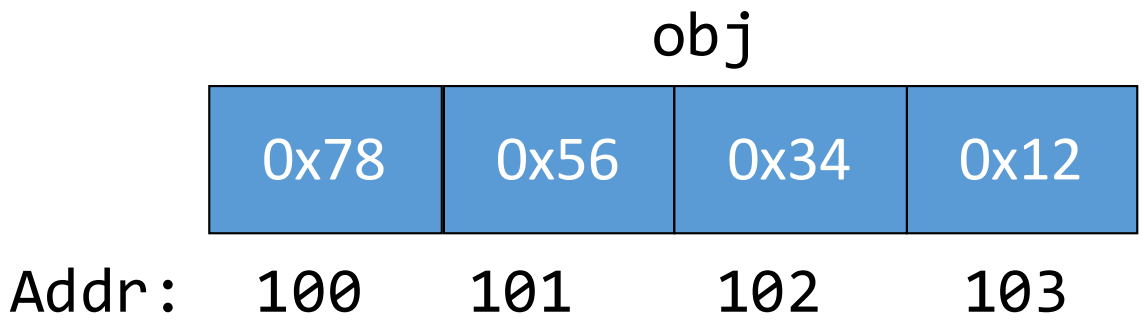
Union – memory layout

```
typedef union {
int i;
char c;
}su;

su obj; //size of obj is 4 bytes (assuming int
occupies 4 bytes)
```



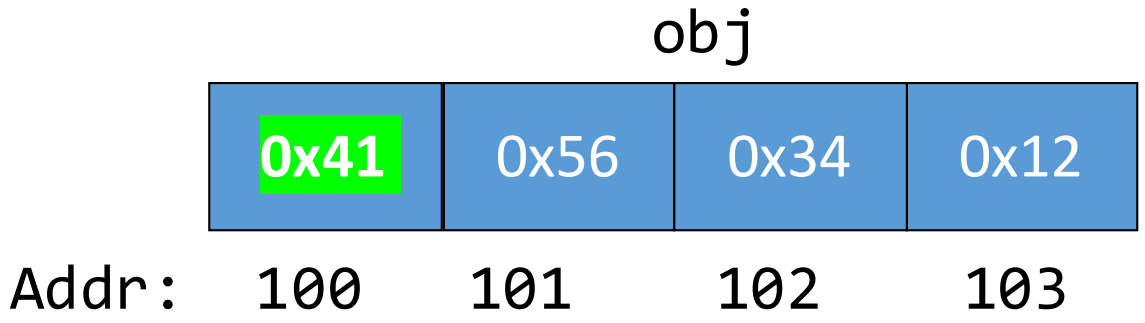
Union – memory layout

```
obj.i = 0x12345678;
```



Union – memory layout

```
obj.c = ‘A’;
```




```
Ascii value of ‘A’ is 65 = 0x41
```

Complex structures

- Can have struct members within (`addr` is a member of `student`, and its type is `struct Addr`)

```
typedef struct {
  int ID;
  char name[MAX_LEN];
  Address addr;
}Student;

typedef struct {
char* st;
int zip;
}Addr;
```

Complex structures

- Accessing members is not different from accessing structure members

```
char* str=malloc(3); //allocate memory on heap
str[0]='E'; str[1]='C'; str[2]='E'; //initialize memory
Student stu1; //defines a variable stu1 of type Student
stu1.addr //accesses the addr member
stu1.addr.st //accesses the st member of Addr type
stu.addr.st = str; //assigns address of the heap memory
                   location to st
```

Complex structures

- Careful while assigning (shallow-copy)

```
char* str=malloc(3); //allocate memory on heap
str[0]='E'; str[1]='C'; str[2]='E'; //initialize memory
Student stu1, stu2;
stu.addr.st = str; //assigns a dynamically
                   allocated char array to st
stu2=stu1 //copies the value of str.NOT what str points to
free(stu1.addr.st) //frees memory allocated to str
stu1.addr.st = NULL; //resets the pointer
```

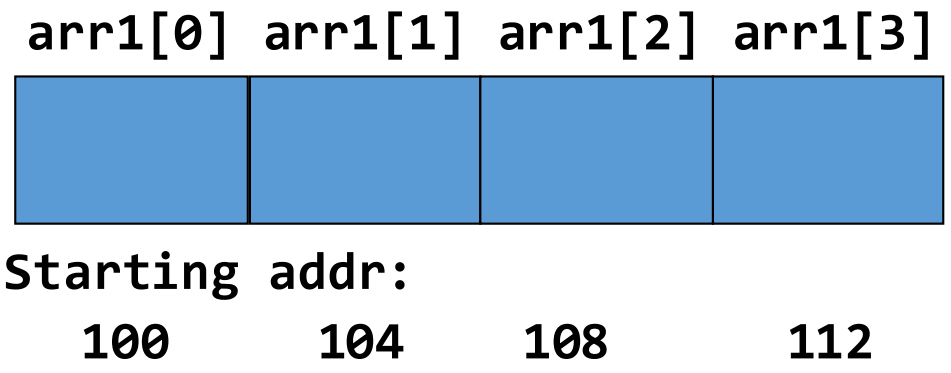
stu2.addr.st still points to memory released

2D Arrays

- 1D array gives us access to *a row* of data
- 2D array gives us access to *multiple rows* of data
  - A 2D array is basically an *array of arrays*

- Consider a fixed-length 1D array

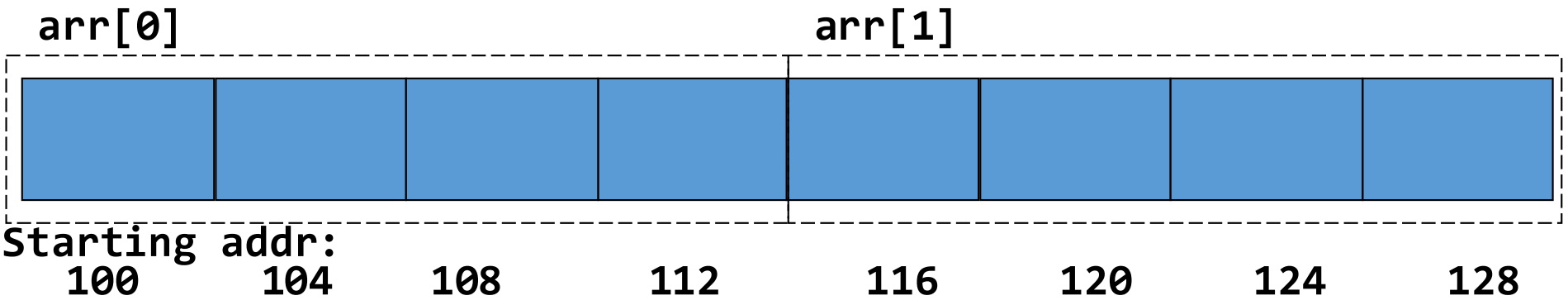
```
int arr1[4];//defines array of 4 elements; every
element is an integer. Reserves contiguous memory to
store 4 integers.
```



2D Arrays (fixed-length)

- Consider a fixed-length 2D array (*array of arrays*). Think:
  array of integers => every element is an `int`
  array of characters => every element is a `char`
  array of array => every element is an *array*
  - Example:

```
int arr[2][4];//defines array of 2 elements; every element is an array of 4 integers. Therefore, reserves contiguous memory to store 8 integers
```



2D Arrays (on heap)

- What if we don't know the length of the array upfront?

  E.g. A line in a file contains number of people riding a bus every trip. Multiple trips happen per day and the number can vary depending on the traffic.

  Day1 numbers: 10 23 45 44
  Day2 numbers: 5 33 38 34 10 4
  Day3 numbers: 9 17 10
  ……
  DayN numbers: 13 15 28 22 26 23 22 21

//we need array arr2D of N elements; every element is an array of M integers. Both N and M vary with every file input.

2D Arrays (on heap)

1. First, we need to create an array `arr2D` of N elements. So, get the number of lines in the input file.

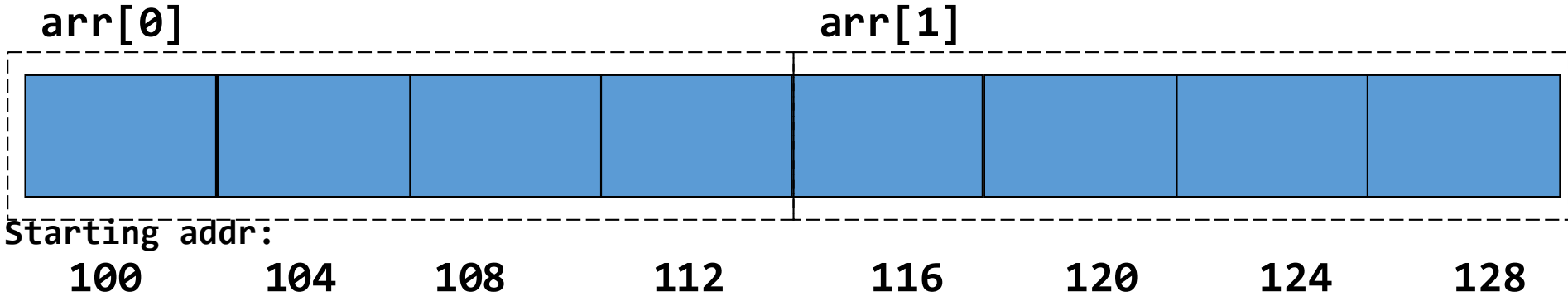
   - But what is the *type* of every element? - array of M elements, where every element is an integer (i.e. every element is an integer array). `int *`

   - What is the type of `arr2D`? (array of array of integers)
     Think:
     type of an integer => `int`
     type of array of integers => `int *`
     (append a * to the type for every occurrence of the term array)
     type of array of array of integers => `int **`

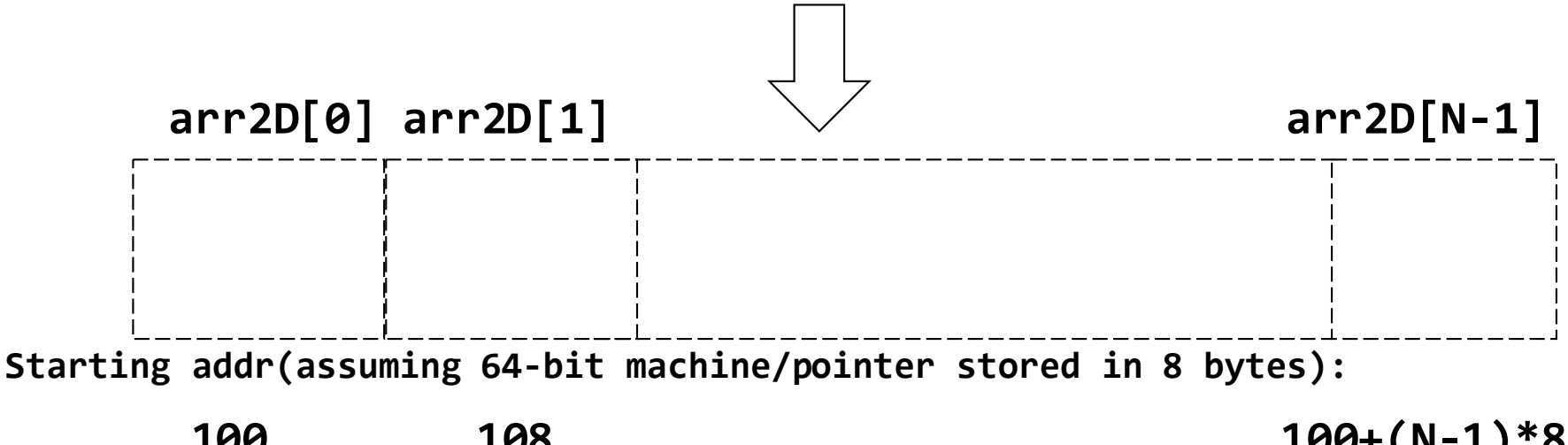
What is the type of arr2D? (int**)

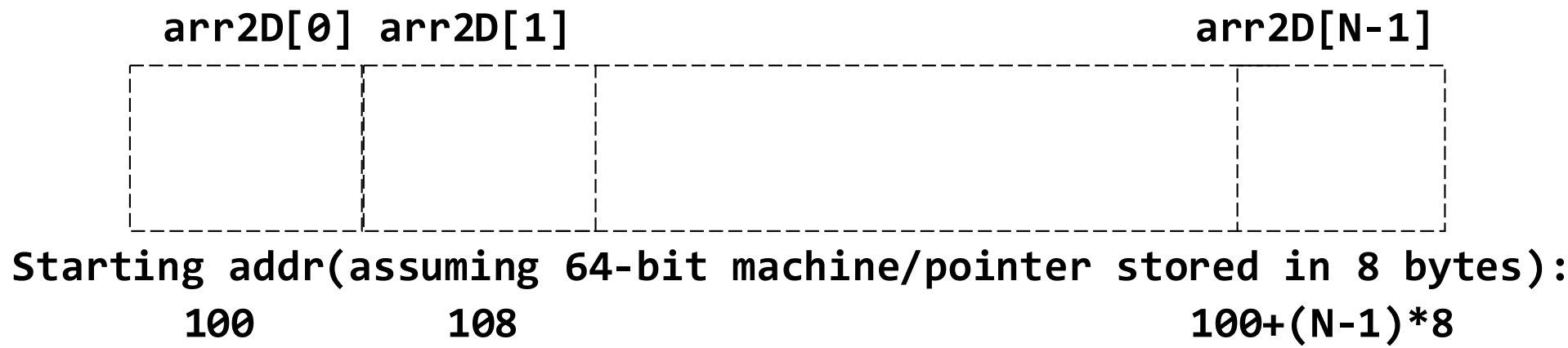
```
int N = GetNumberOfLinesFromFile(argv[1]);

int** arr2D = malloc(sizeof(int *) * N)
```

Recall boxes with dashed lines in `int arr[2][4];`



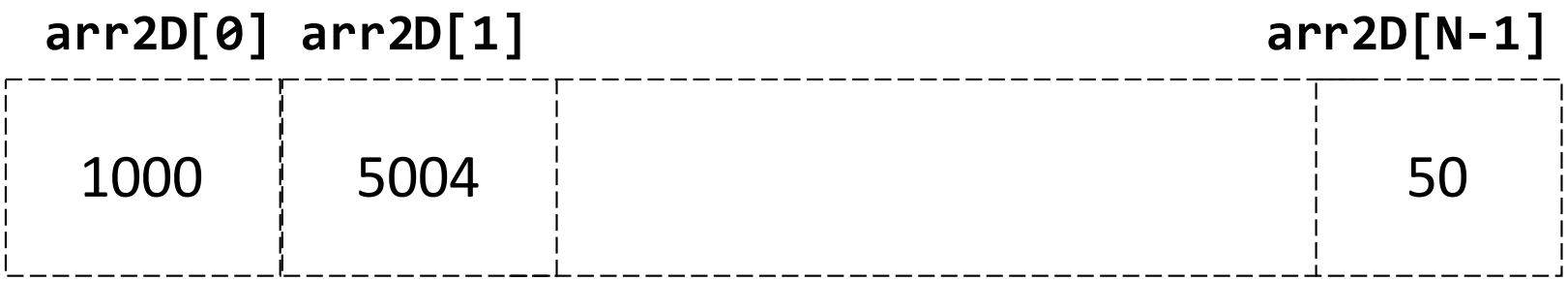

```
int N = GetNumberOfLinesFromFile(argv[1]);
int** arr2D = malloc(sizeof(int *) * N)
```





2. arr2D[0], arr2D[1], etc. are not initialized. Currently they hold random addresses. How do we initialize them?

```
for(int i=0;i<N;i++) {
   char* line = ReadLineFromFile(argv[1]);
   int M = GetNumberOfIntegersPerLine(line);
   arr2D[i] = malloc(sizeof(int) * M)
}
```

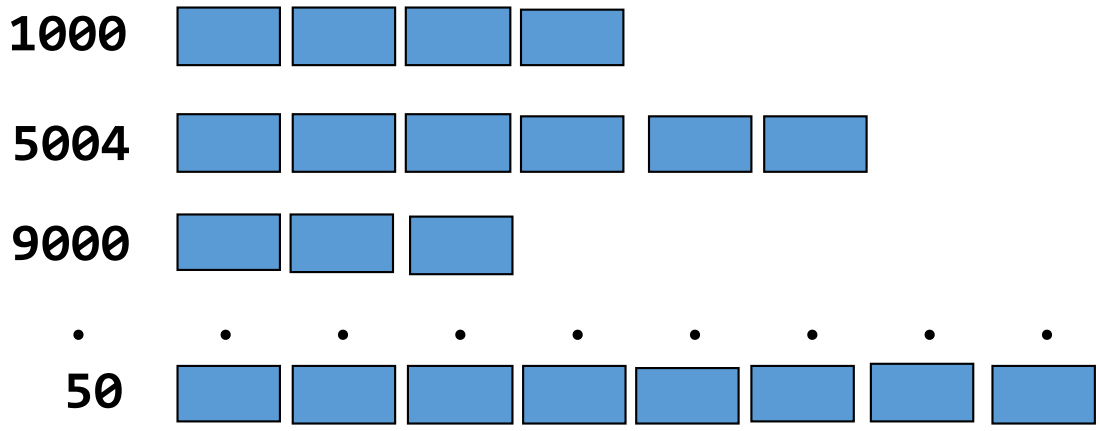


Starting addr(assuming 64-bit machine/pointer stored in 8 bytes):

100 108 100+(N-1)*8

```
for(int i=0;i<N;i++) {
   char* line = ReadLineFromFile(argv[1]);
   int M = GetNumberOfIntegersPerLine(line);
   arr2D[i] = malloc(sizeof(int) * M)
}
```

Starting addr:



2D Arrays (on heap)

2. Now we need to initialize each of the array elements (in the second dimension)

Summary:

Creation: 2-steps

Initializing: 2-steps

Freeing: 2-steps

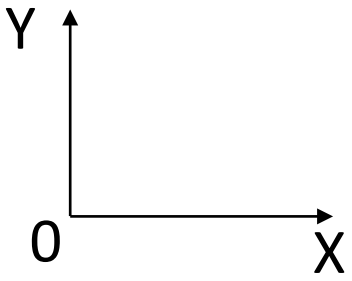
```
for(int i=0;i<N;i++)
    free(arr2D[i]); //frees memory at 1000, 5004, etc.
free(arr2D);//frees memory at 100
```

2D Arrays (trivia)

- Notation used to refer to elements different from cartesian coordinates

arr2D[0][0] accesses 1st row, 1st element
arr2D[0][1] accesses 1st row, 2nd element
arr2D[1][1] accesses 2nd row, 2nd element
arr2D[N][M] accesses N+1th row, M+1th element

- Cartesian:



- (M,N) = move M along X axis, N along Y axis
- In 2D array notation (M, N) means move to M+1th row (along Y axis), to N+1th column (along X axis)!

- From the previous bus trip data, what if we wanted to:

```
Day1 numbers:  10  23  45  44
Day2 numbers:   5  33  38  34  10  4
Day3 numbers:   9  17  10
 ……
DayN numbers: 13  15  28  22  26  23  22  21
```

  - Drop certain days as we analyzed arr2D?
  - Add more days to (read from another file) to arr2D ?

i.e.

modify arr2D as program executes?

Dynamic Data Structures

- We use *dynamic data structures*
  - Allocate more space as we realize that we need to store more data
  - Free up space when we realize that we are storing less data
- Example:
  - Linked Lists, Trees, Stacks, Queues etc.

Linked Lists

- Most basic dynamic data structure
- Create a linked set of *structures* (`struct` objects)
  - Each structure holds a piece of data, and a *pointer* to the *next* structure in the list, which holds the next piece of data

How can we create such a structure?

Linked Lists

- Use recursive structure definition
  - Pointer from *within* the structure to another structure *of the same type*
  - Example: (structure holds integer data)

```
typedef struct Node {
        int val;
      struct Node* next;
}Node;
```

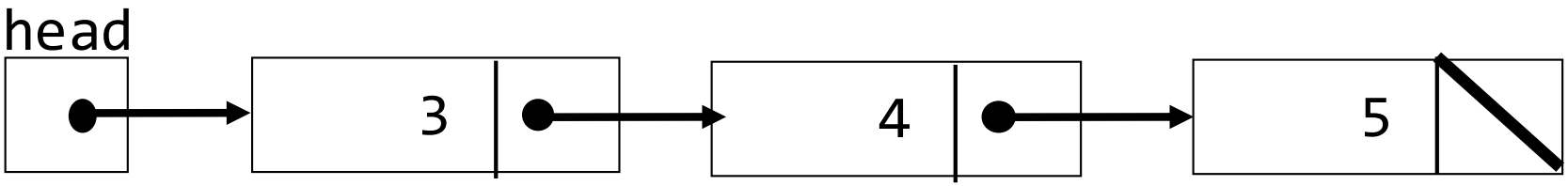
  - Note the ‘`struct Node`’ as part of the type for member `next`

Linked Lists

- Graphical representation of Node

- Creating a list of integers:

```
Node* head = malloc(sizeof(Node));
head->val=3;
head->next = malloc(sizeof(Node));
head->next->val=4;
head->next->next = malloc(sizeof(Node));
head->next->next->val=4;
Head->next->next->next = NULL;
```

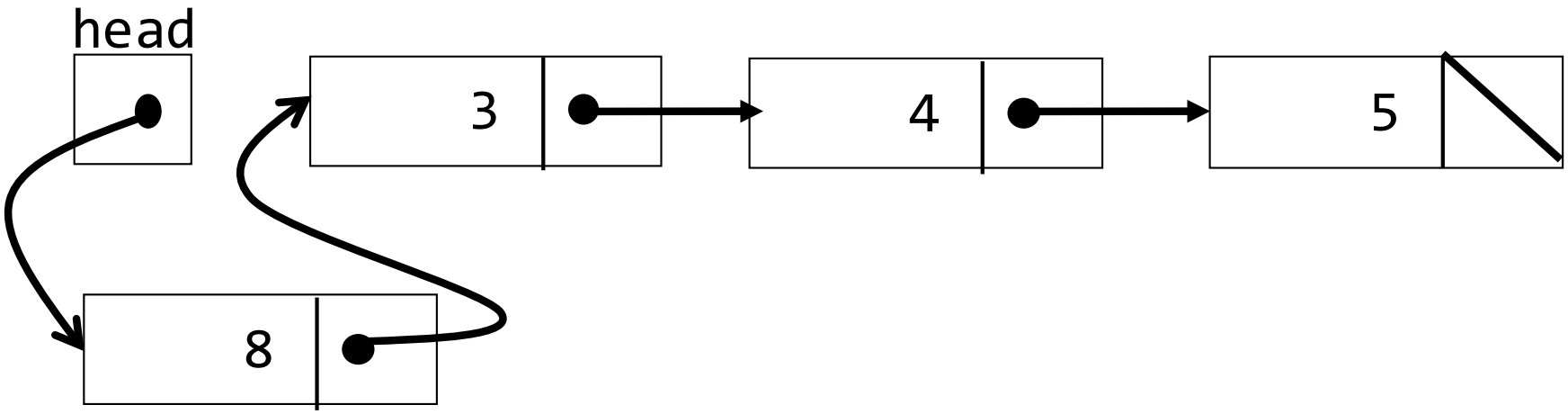


- head is a `Node *` just like `next:` use same sized box to represent head
- Using head, we can get to any node in the list
  - Just follow the `next` pointers
- `Next` field of last node is `NULL.` So, represent it with a slash

Linked Lists (updating the list)

- Add a new number

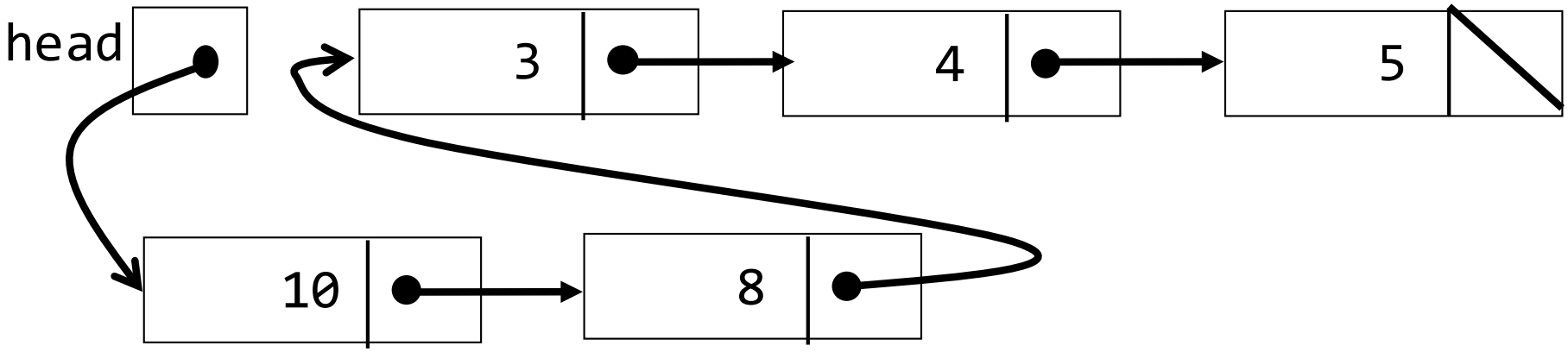
```
Node* newNode = malloc(sizeof(Node));
newNode->val=8;
newNode->next = head;
head = newNode;
```



Linked Lists (updating the list)

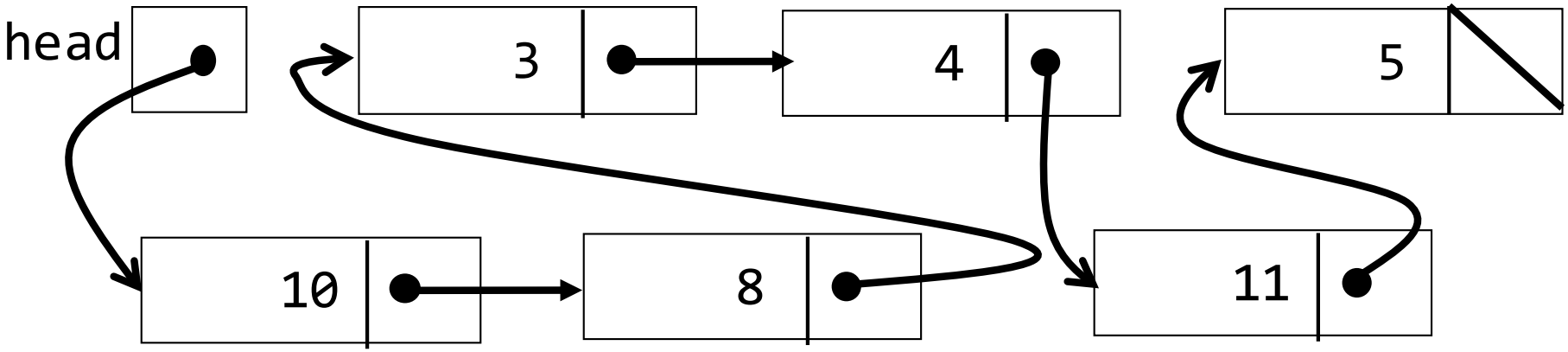
- Add a new number:

```
void insert(Node** loc, int val) {
    Node* newNode = malloc(sizeof(Node));
    newNode->val=val;
    newNode->next = *loc;
    *loc = newNode;
}
insert(&head, 10);
```



Linked Lists (updating the list)

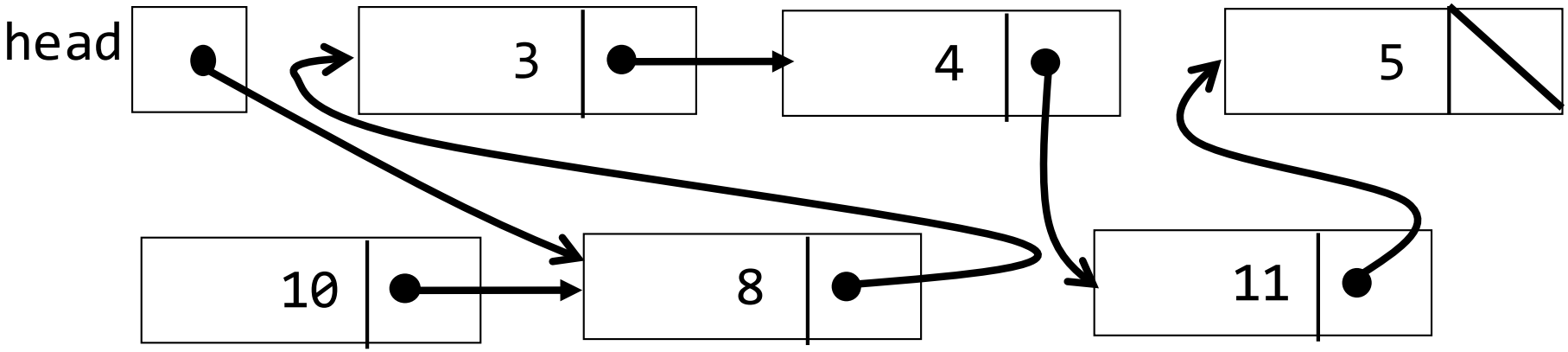
```
Node* cur=head->next->next->next->next;//points to 4
insert(&head, 11);
```



Linked Lists (removing from list)

Removing the first node:

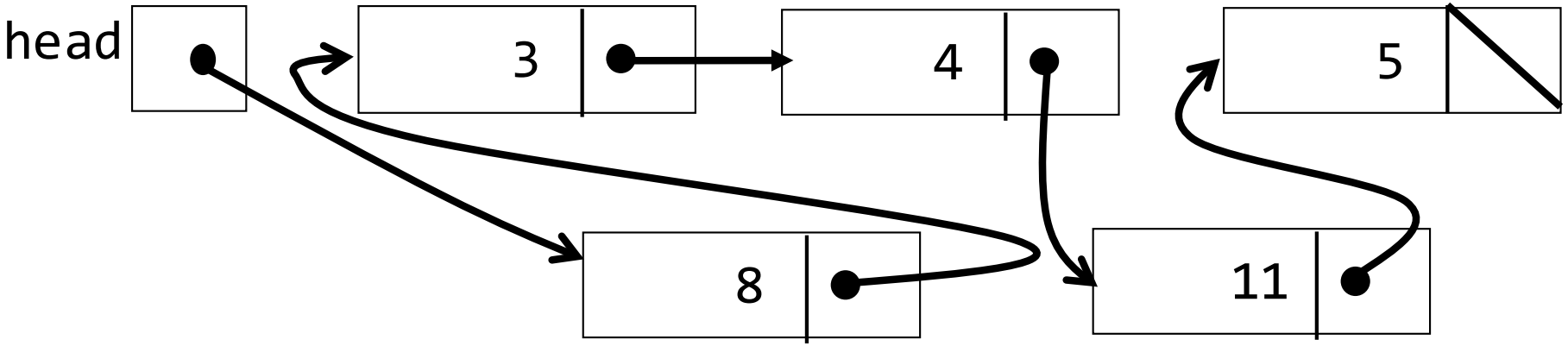
```
head = head->next;
```



What is the problem here?

Linked Lists (removing from list)

```
Node* toDelete = head;
head = head->next;
free(toDelete);
```



Linked Lists (removing from list)

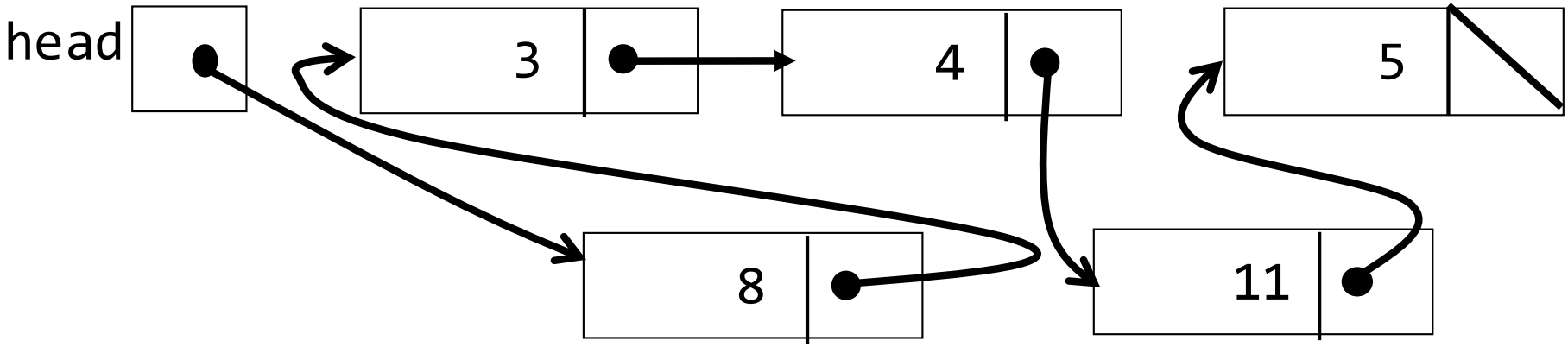
Create a function for removal

```
void remove(Node** loc) {
      Node* toDelete = *loc;
      *loc = (*loc)->next;
      free(toDelete);
}
```

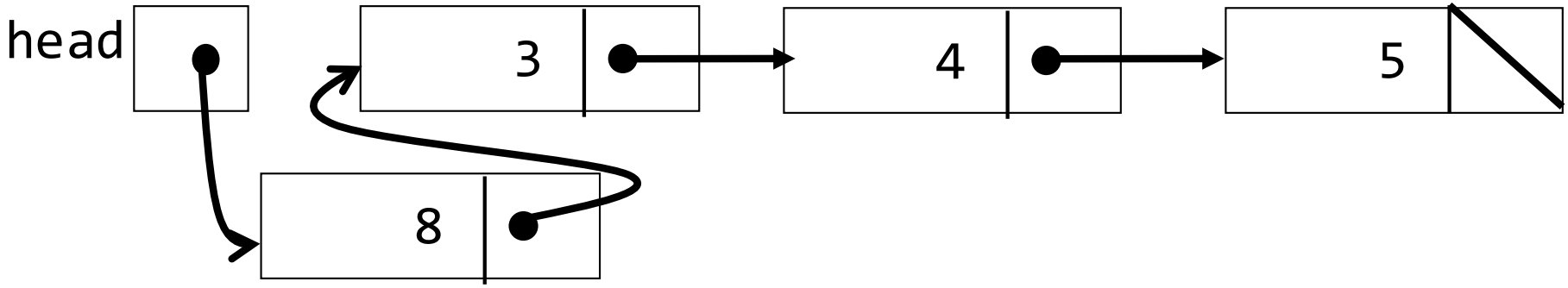
- We can pass any address of a pointer to `Node`.
- To remove a node from the list, pass the address of the next pointer that *points to* that node.

Linked Lists (removing from list)

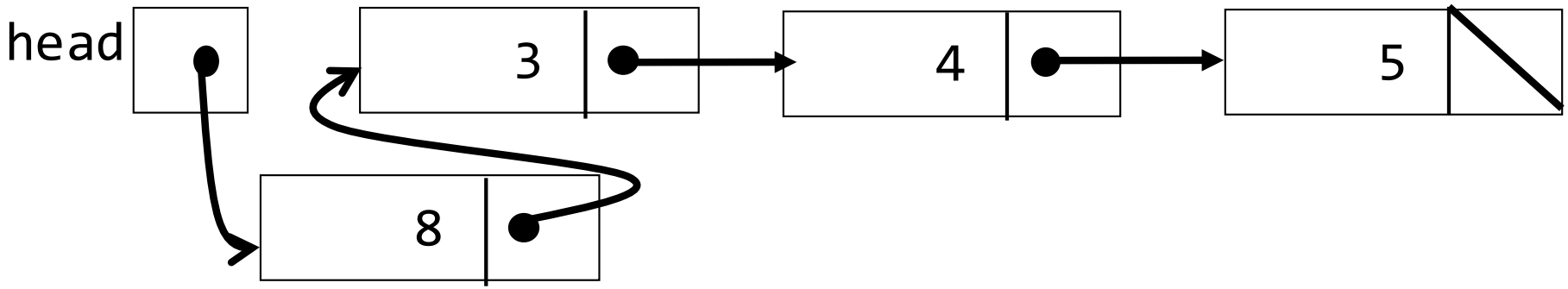
Example:




```
Node* del = head->next->next; //points to 4
remove(&(del->next)); //removes 11
```



Exercise (removing from list)




```
void remove(Node** loc) {
      Node* toDelete = *loc;
      *loc = (*loc)->next;
      free(toDelete);
}
Node* del = head->next->next->next; //points to 5
remove(&(del->next)); //What happens here?
```

Linked Lists (searching a list)

1. Step through each node in the list
2. Check if `val` in each node matches key
3. Return `true` if matches, `false` if no match found till end of list

```
bool contains(Node* head, int key) {
      Node* cur = head;
      while(cur != NULL) {
            if(cur->val == key) return true;
            cur = cur->next;
      }
      return false;
}
```

List manipulation

- Given a key, delete that node:
- 1) Find the key. If found return the address of the next pointer that points to that node. If not, return address of last next pointer

```
Node** findEq(Node** loc, int key) {
      while((*loc) != NULL) {
            if((*loc)->val == key) return loc;
            loc = &((*loc)->next);
      }
      return loc;
}
```

List manipulation contd..

2) pass the returned address from `findEq` to `remove`.

```
Node** toRemove = findEq(&head, 8);
if((*toRemove) != NULL)
      remove(toRemove);
```