alex-levesque.com

ELEC 278 - Lecture Notes
FUNDAMENTALS OF INFORMATION STRUCTURES

Prof. Kexue Zhang • Fall 2025 • Queen's University

Contents

#ECE

1 Data Structures Overview

[[09-03 Data Structure]] [[09-04 Overview of C]] [[09-08 Structures]] [[09-10 Overview of C contd 2]] [[09-11 Linked Lists]] [[09-15 Linked Lists contd]] [[09-17 Double Linked Lists and Dynamic Arrays]] [[09-18 Double Linked Lists and Dynamic Arrays contd]] [[09-22 Stack]] [[09-24 Queue]] [[09-25 Recursion]] [[09-29 Recursion contd]] [[10-01 Algorithm Analysis]] [[10-02 Big Oh Notation]] [[10-08 Trees]]

#ECE

2 What is a Data Structure

A data structure is a way to store and organize data in order to facilitate access and modification

It concerns the **representation, manipulation, and efficient management of data**. Use cases involve large databases and internet indexing services.

Efficient data structures are key to designing efficient algorithms.

Each data structure supports one or more algorithms for the **operations**, that are insert, delete, search, and modify

Conversely, an algorithm is a step by step procedure in performing a task

#ECE

3 Overview of C

Incorrect swap function

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = tmp;
    }

int x = 1;
int y = 2;
swap(x,y);
```

This function is swapping its own copy of the data, and these changes are not reflected outside the function Pointers are necessary to affect the original variables

3.1 Pointers

```
int *nptr = &total;
```

This pointer is pointing to the memory address of the total variable

The & operator returns or takes the address of a variable

Usage:

```
int total = 5;
float speed = 2.3;
int numbers[4];
int *nptr = &total;
*nptr = 12; # this changes 5 to 12
```

The * operator uses or dereferences the value of the pointer

3.2 Correct swap function

```
void swap(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = *a;
    }
int x = 1;
int y = 2;
swap(&x, &y); # x and y are changed
```

Pointer values are copied, and pointers are used to access original values

3.3 Indexing Pointers

Neglecting [] of an array calls its first value (at index 0)

```
int numbers[4];
int *nptr = numbers;
```

Notice, no & for pointing to arrays

You can add or subtract integers from pointers. When you do this, the pointers value is scaled by the size of its referenced type (e.g. nptr + 2 is equivalent to adding 8 bytes if pointing to int since each int is 4 bytes)

You can access array elements using pointer arithmetic:

```
*(nptr+2)=10; # sets the third element of the array to 10
```

3.4 Global and Local Variables

Global variables are always initialized to zero

#ECE

4 Structures

A data structure is a way to store and organize data in order to facilitate access and modification

4.1 C Structures

This groups items of possibly different types into a single type

```
struct structure_name{
    data_type member_name1;
    data_type member_name2;
    };
```

The items in the structure are called **members** or **fields**

We can define structure variables as

```
struct structure_name variable_name1;
```

or at the same time:

```
struct structure_name{
    data_type member_name1;
    data_type member_name2;
    } variable_name1;
```

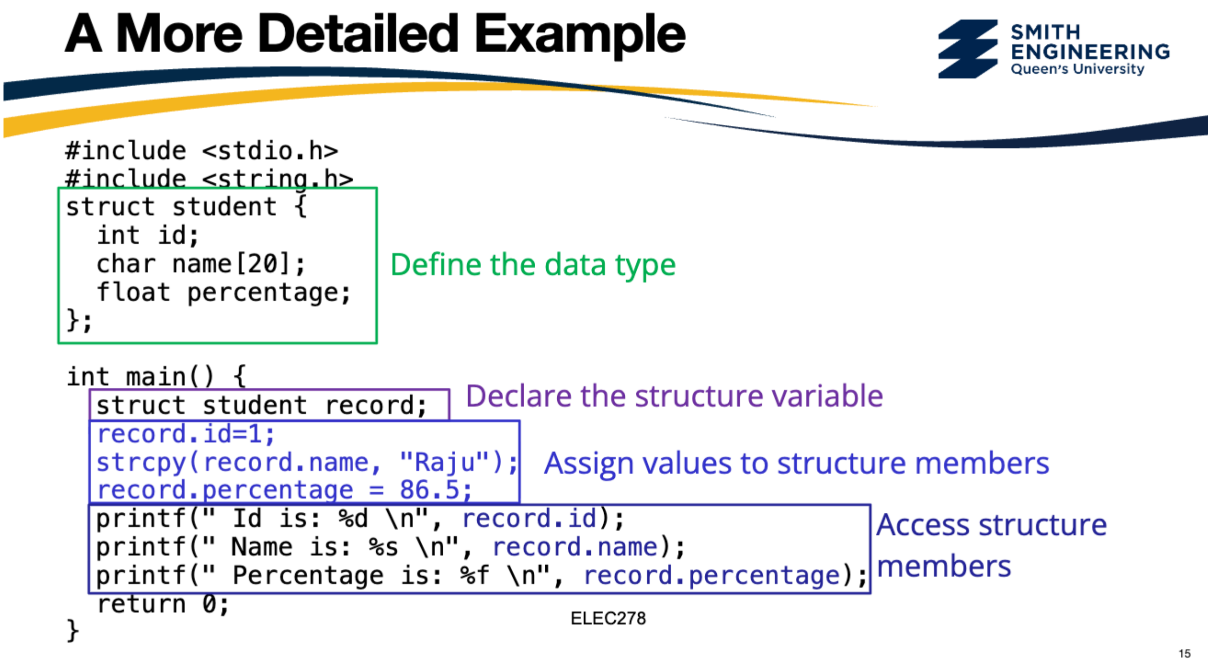
Where *structure_name* is the defined structure of data, and *variable_name* is an instance of the struct

Use the dot (.) operator to access members

```
var_name.marital_status = 4;
```

and initialize individual fields with the dot notation:

```
struct Student student1={.name = "Alice", .section = 'A'};
```



A More Detailed Example

```
#include <stdio.h>
#include <string.h>
struct student {
  int id;
  char name[20];
  float percentage;
};

int main() {
  struct student record;
  record.id=1;
  strcpy(record.name, "Raju");
  record.percentage = 86.5;
  printf(" Id is: %d \n", record.id);
  printf(" Name is: %s \n", record.name);
  printf(" Percentage is: %f \n", record.percentage);
  return 0;
}
```

Define the data type
Declare the structure variable
Assign values to structure members
Access structure members
SMITH ENGINEERING Queen's University
ELEC278
15

4.2 Typedef

A typedef looks like a variable definition, but defines a new name for an existing type. This can improve code readability.

```
typedef int studentNumberType;
studentNumberType studentNumber1;
```

Here, studentNumberType becomes an alias for int, so studentNumber1 is actually an int but is made clearer.

```
typedef struct {
    char name[50];
    int class;
    char section;
    } Student;
Student s1, s2;
```

4.3 Nested Structures

If we typedef a struct, and use it in another structure, one of the members of a structure is itself another structure

```
typedef struct {
    int imag;
    float real;
} complex;

struct number {
    int flags;
    complex phase;
} num1, num2;
```

Where *phase* is a *struct*

4.4 C Enumerated Types

This is a way to create a user-defined type consisting of a set of named integer constants

```
enum colors {RED, GREEN, BLUE}
```

You can also change the starting value by:

```
enum week {Monday = 1; ...} # Sunday is 0
```

We can also combine enums with typedef

4.5 More Structs

Structs can be used as the base type of an array

```
typedef struct {
    int x;
    int y;
} point;

point vertexes[100];

vertexes[4].x=23;
vertexes[4].x=18;
```

We can also point to members within a struct

```
typedef struct {
    int x;
    int y;
} point;

point a = {23,18};
point *b = &a;
(*b).x=34;
```

We can also return structs as a result of a function

4.6 Pointers to pointers

Pointers to pointers are used to reference the address of another pointer

```
void allocateInt(int **p) {
    *p = malloc(sizeof(int))
}
```

or

```
void swap(point **a, point **b){
    point = *tmp;
    tmp = *a;
    *a = *b;
    *b = *tmp;
}
```

4.7 Dynamic Memory

So far, memory was either global or local. Now we add **Heap memory**, to request memory at runtime

```
tax_info *bob = (tax_info*) malloc(sizeof(tax_info));
bab->martial_status = single;
```

The arrow operator is shorthand for (*ptr).field

4.8 Arrays of pointers

Instead of storing 100 structs (3200 bytes), you can store 100 pointers (800 bytes) and allocate each struct individually:

```
tax_info *employees[100];
employees[0] = malloc(sizeof(tax_info));
```

4.9 Dynamic strings

To duplicate a string dynamically:

```
char *s1 = "Hello";
char *s2 = strdup(s1); # strdup is like copying into another string
```

4.10 Function pointers

Since functions also live in memory, we can point to them too:

```c
int sum(int a, int b) { return a+b; }
int (*func_ptr)(int,int) = sum;
printf("%d\n", func_ptr(2,3));
```

#ECE

5 Linked Lists

Use structure to group array and count of number elements in the array * Arrays are declared with a maximum length

Linked lists provide an ordered collection that grows with the number of data items * each element is a node with data and a pointer to the next element * Head points to the first element

5.1 Linked Lists

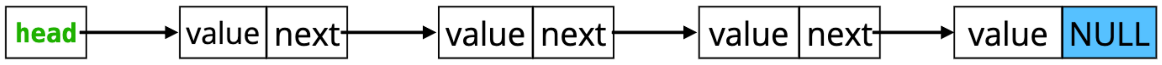
Linked lists are a means of dynamically allocating an ordered collection of elements

Why?: Arrays are rigid. If you dont know how many items youll need, arrays can overflow. Linked lists solve this with dynamic allocation.

Each element (**a node**) stores: * Data * Pointer to the next node

The list is ended with a *NULL* ptr

```c
struct node {
    int value;
    struct node *next;
};
struct node *head; // points to first node
```



5.2 Building a Linked List

To begin building a linked list, memory is first allocated for the head node using `malloc`, and its value field is set to `14`

```c
head = (struct node *) malloc (sizeof(struct node));
head -> value = 14;
```

Next, another node is added by allocating memory for `head->next`, and its value is set to `92`

```c
head -> next = (struct node *) malloc (sizeof(struct node));
head -> next -> value = 92;
```

After allocating each node, it is important to check if memory allocation succeeded; if `head` is `NULL`, the program prints failed and exits

```
if (head == NULL) {
    printf("failed");
    exit(1);
};
```

We can use linked lists with more complex datatypes:

Example:

```
typedef struct {
    float x;
    float y;
    int alt;
} aircraftPos;
```

We can build a node struct, which is the building block of the linked list, where each node contains a value of type aircraftPos and a pointer to the next node on the list

```
typedef struct node {
    aircraftPos value;
    struct node *next;
} node;
```

5.3 Traversing a linked list

Start function

```
void printList(struct node *head) {
```

To traverse a linked list, you begin by pointing a temporary node `tmp` to the head, then print an opening bracket to start the list display

```
struct node *tmp = head;
printf("[ ");
```

The traversal continues in a loop while `tmp` is not `NULL`; this ensures you stop once you reach the end of the list

```
while (tmp != NULL) {
```

Inside the loop, you first access the current nodes value by printing it, and then move forward by updating `tmp` to its `next` pointer

```
    printf("%d, ", tmp->value);
    tmp = tmp->next;
}
```

Finally, you print a closing bracket and a newline to finish displaying the entire list

```
print("]\n");
```

5.4 Adding a new node to an empty list

To simplify creating nodes, we can define a helper function `newNode` that allocates memory with `malloc`, assigns the given value, initializes the `next` pointer to `NULL`, and returns the new node

```
node *newNode(int value){
    node *tmp = (node *)malloc(sizeof(node))
    tmp -> value = value;
    tmp -> next = NULL;
    return tmp;
}
```

To add a new node to an empty list, we call `newNode` to create a node with value `14`, and if the head is `NULL` (meaning the list is empty), we set `head` to point to this new node

```
new = newNode(14);
if (head != NULL) {
    head = new;
}
```

We can also insert at the start of an existing list by creating a new node, setting its `next` pointer to the current head, and then updating head to point to this new node

```
node *new = newNode(12);
new->next = *h;
*h = new;
```

This insertion-at-front logic can also be written as a reusable function, `insertFirst`, which takes a double pointer to the head, creates a new node, links it before the current head, and updates the head pointer.

```
void insertFirst(node **h, int data){
    node *new = newNode(data);
    new -> next = *h;
    *h = new;
}

insertFirst(&head,14);
```

5.5 Appending to an existing list

Start function

```
void appendList(node **h, int data){
  node *new = newNode(data);
```

Cover empty list case:

```
if (*h == NULL) { *h = new; return; }
```

To append to an existing linked list, we first set a pointer `tail` to the head of the list, and then create a new node with the desired value using `newNode`

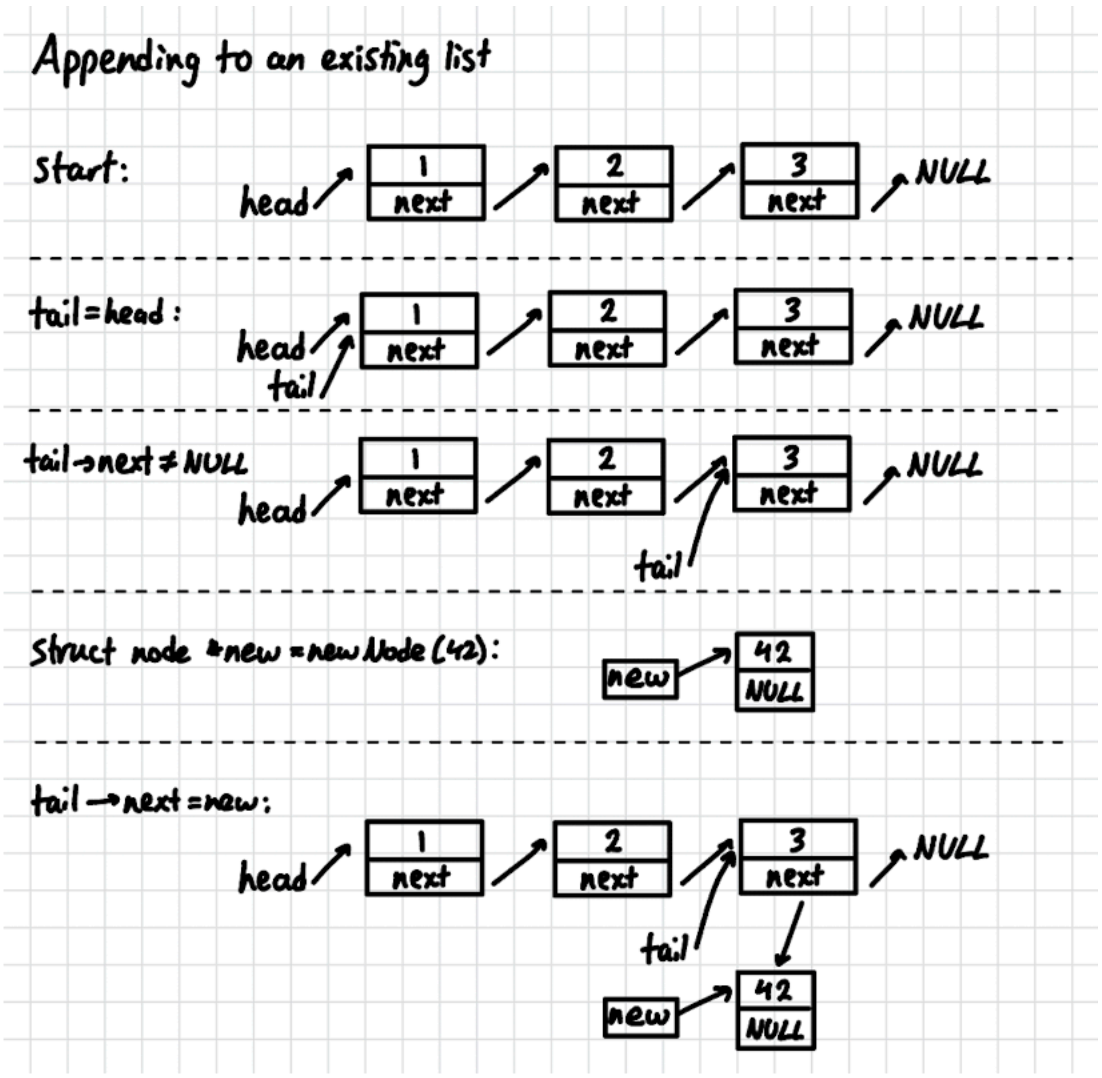
```
node *tail = *h; // start at the head
struct node *new = newNode(42);
```

Next, we traverse the list by moving `tail` forward until we reach the last node, which is identified when `tail->next` becomes `NULL`

```
while (tail -> next != NULL) {
    tail = tail -> next; // traverse to the last node
    }
```

Finally, once at the last node, we attach the new node by setting `tail->next` to point to it, completing the append operation

```
tail -> next = new; // link the new node at the end
```



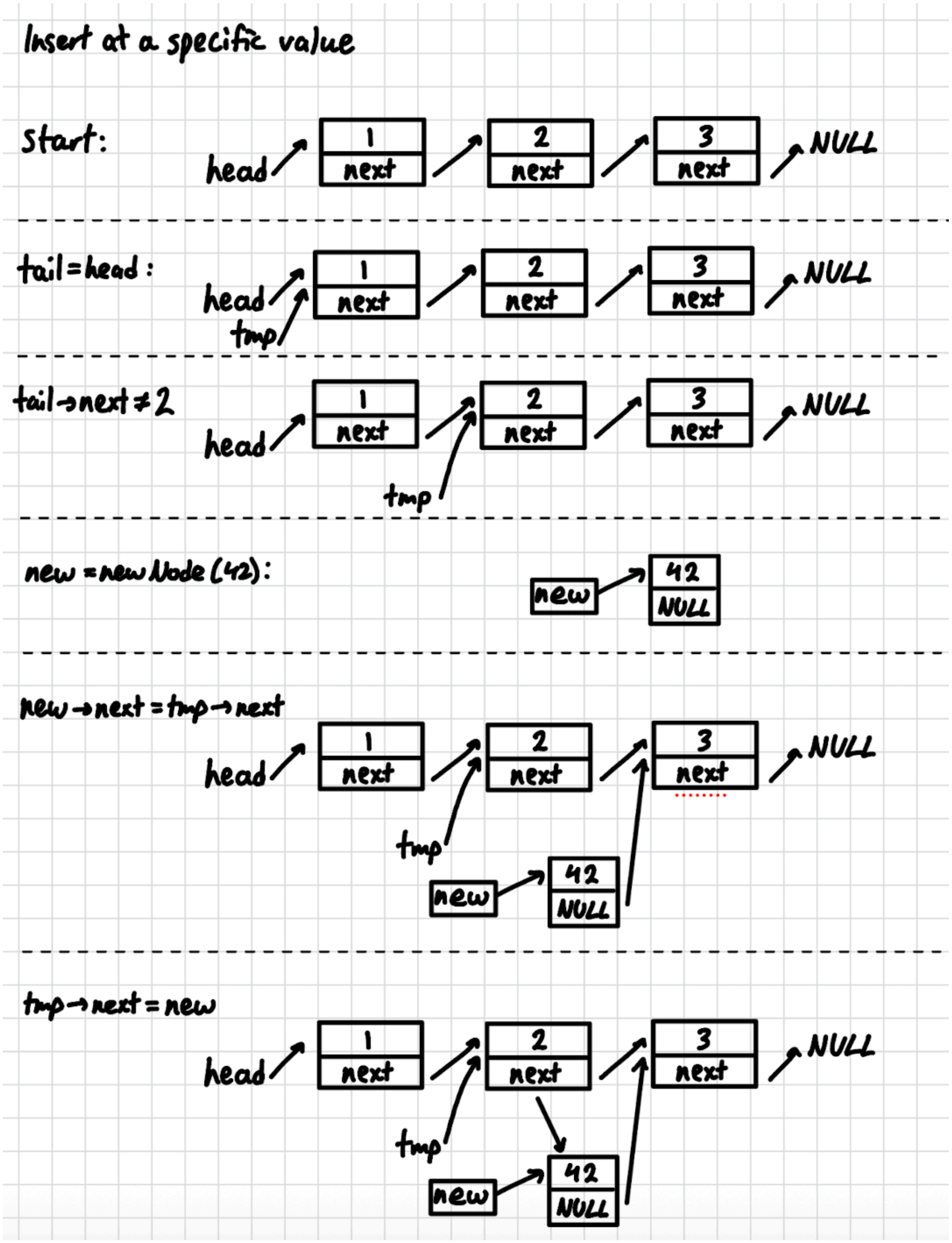
5.6 Insert a Value in the Middle of a List (after a specific value)

To insert a new node after a specific value, we first create a temporary pointer `tmp` starting at the head, and then traverse forward while `tmp` is not `NULL` and the current nodes value is not equal to `92`

```
struct node *tmp = head; // start at head
while (tmp!=NULL && tmp->value!= 92){
  tmp = tmp->next;  // move to the next node
}
```

If the loop finds a node with value `92`, then `tmp` points to the node. At this point, we create a new node with value `42`, set its `next` pointer to the node after `tmp`, and then adjust `tmp->next` so that it links to the new node

```
if (tmp != NULL){
  new = newNode(42); // create node with value 42
  new -> next = tmp -> next; // link new node to the node after tmp
  tmp -> next = new; // insert new node right after tmp
}
```



5.7 Insert Sorted Function

The function starts by creating a new node with the given value using `newNode`

```
node *new = newNode(value);
```

If the list is empty, the new node becomes the head, and the function immediately returns

```
if (*head == NULL){              // Case 1: Empty list
    *head = new;
    return;
}
```

If the new value is smaller than the current heads value, the new node is inserted at the front, and head is updated

```
if ((*head)->value > value){   // Case 2: Insert before the first
 ↪  node
    new->next = *head;
    *head = new;
    return;
}
```

Otherwise, the function traverses the list using a pointer `tmp`, always looking one node ahead. If it finds that the next nodes value is larger than the new value, it inserts the new node between `tmp` and `tmp->next`

```
node * tmp = *head;
while (tmp->next != NULL){               // Traverse the list
    if (tmp->next->value > value){       // Case 3: Insert in middle
        new->next = tmp->next;           // Point new node to next node
        tmp->next = new;                 // Link previous node to new
 ↪  node
        return;
    }
    tmp = tmp->next;                     // Advance to next node
}
```

If no larger value is found, the function appends the new node at the end of the list

```
tmp->next = new; // Case 4: Insert at the end
```

5.8 Removing Elements

Often, we want to remove the first node, a specific node, or the last node

Delete First Node

The function `removeFirst` begins by saving a pointer `tmp` to the current head of the list and initializing an integer variable `value` to `0`, which acts as a default return value

```
struct node *tmp = *head;    // Save pointer to current head
int value = 0;               // Default return value if list is empty
```

It then checks if the list is non-empty, if so, the head pointer is advanced to the second node, the value of the original head is saved into `value`, and the memory allocated to the old head node is freed

```
if (*head != NULL){
    *head = (*head)->next;  // Move head to next node
    value = tmp->value;     // Save the value of the old head
    free(tmp);              // Free memory of the old head
}
```

Finally, the function returns the integer `value`, which is either the removed nodes value or `0` if the list was empty

```
return value'
```

Delete Specific Value

The function `deleteValue` starts by checking if the head of the list is `NULL`, and if so, it immediately returns because there is nothing to delete

```
if (*head == NULL) return;
```

Next, it handles the special case where the first node contains the target value: it saves a pointer to the head in `tmp`, updates `head` to the second node, frees the old head, and then returns.

```
if ((*head)->value == val) {
  node * tmp = *head;
  *head = (*head)->next;
  free(tmp);
  return;
}
```

If the value is not in the head, the function creates a pointer `prev` starting at the head, and then iterates while `prev->next` is not `NULL`

```
node * prev = *head;
while (prev -> next != NULL) {
```

Inside the loop, it checks if the next node holds the target value; if so, it saves that node in `tmp`, bypasses it by updating `prev->next`, frees the node, and returns.

```
  if (prev -> next -> value == val){
    node * tmp = prev -> next;
    prev -> next = tmp -> next;
    free(tmp);
    return;
  }
```

If the value was not found, the loop moves `prev` to the next node and continues searching until the end of the list.

```
prev = prev->next;
```

#ECE

6 Double Linked Lists and Dynamic Arrays

6.1 Double Linked List

A single linked list has a single entry point (head)

Idea: Each node has `next` and `prev`, plus you often keep `head` and `tail` pointers so you can traverse both ways.

Hence a Double Linked List, where one linked list goes forward and one goes backwards

```
typedef struct node {
  int value;              // payload
  struct node * next;     // forward link
  struct node * prev;     // backward link
} node;

node * head;              // first node pointer
node * tail;             // last node pointer
```

6.2 Dynamic Arrays

Idea: Wrap an array with a count field so you know how many elements are live

```
typedef struct {
    int *array;
    size_t capacity; // available space
    size_t length; // nmber of elements stored in the array
} Array;
```

Initialization

Allocate an initial buffer, starting at length 0 and capacity at initial size

```
void initArray(Array *a, int initialSize){
  a->array = (int *)malloc(initialSize * sizeof(int)); // allocate
↪  buffer
  a->length = 0;                   // nothing used yet
  a->capacity = initialSize;       // remember capacity
}
```

Append an Element to the Array

```
void addToArray(Array *a, int element){
```

The function begins by checking if the array is already full. It compares the number of used entries (`length`) to the total slots available (`capacity`). If they are equal, the array must grow.

```
if (a->length == a->capacity){
```

In that case, it doubles the capacity and resizes the buffer with `realloc`, ensuring that all existing elements are preserved while making space for new ones

```
a->capacity *= 2;
a->array = (int*)realloc(a->array, a->capacity * sizeof(int));
}
```

After ensuring enough capacity, the function places the new element at the current end of the array (`a->array[a->length]`), then increments `length` so the array correctly reflects its new size

```
a->array[a->length++] = element;
```

And free the dynamic array

```
void freeArray(Array *a){
  free(a->array);                   // release heap buffer
  a->array = NULL;              // null out dangling pointer
  a->capacity = a->length = 0;     // reset counters
}
```

Note: Arrays are contiguous memory. Inserting an element into the memory of the list means moving the remaining elements up one position

We use `memove` to copy `n` bytes from the memory pointed to by `src`, to the memory pointed to by `dst`. This function understands memory blocks overlapping

```
void * memove(void *dst, const void *src, size_t n)
```

6.3 Inserting an Element in the Middle

```
bool insertAt(Array *a, size_t index, int value){
```

The function begins by validating the requested index. If the index is greater than `length`, the function returns `false`

```
if (index > a->length)    // inserting past the end is invalid
  return false;
```

Next, it checks if the array is already full. If `length == capacity`, the capacity is doubled, and the buffer is resized using `realloc`. This ensures there is enough space for the new element

```
if (a->length == a->capacity){
  a->capacity *= 2;
  a->array = realloc(a->array, a->capacity * sizeof(int));
}
```

To make room for the new element, all elements from `index` up to the last used position are shifted one slot to the right. This is efficiently handled with `memmove`, which safely copies overlapping memory regions

```
memmove(&a->array[index + 1],       // shift destination
        &a->array[index],           // shift source
        (a->length - index) * sizeof(int)); // number of bytes to
        ↪  shift
```

Finally, the new value is written into the open slot at `index`, and `length` is incremented to account for the insertion. The function returns `true` to indicate success

```
a->array[index] = value;
a->length++;
return true;
```

6.4 Deleting from the middle

The function begins by validating the index.

```
if (index >= a->length)    // invalid index (out of range)
  return false;
```

If the index is valid, the function reduces the logical size of the array by decrementing `length`. This means the element at the given index will effectively be removed

```
a->length--;
```

To fill the gap, all elements after the removed one are shifted left by one slot. `memmove` is used here because it safely handles overlapping regions of memory

```
memmove(&a->array[index],         // overwrite deleted slot
        &a->array[index + 1],     // start from next element
        (a->length - index) * sizeof(int)); // shift the suffix left
```

After the shift, the function optionally applies a shrink policy. If the number of elements is at or below one-quarter of the current capacity, the capacity is halved and the buffer resized with `realloc`

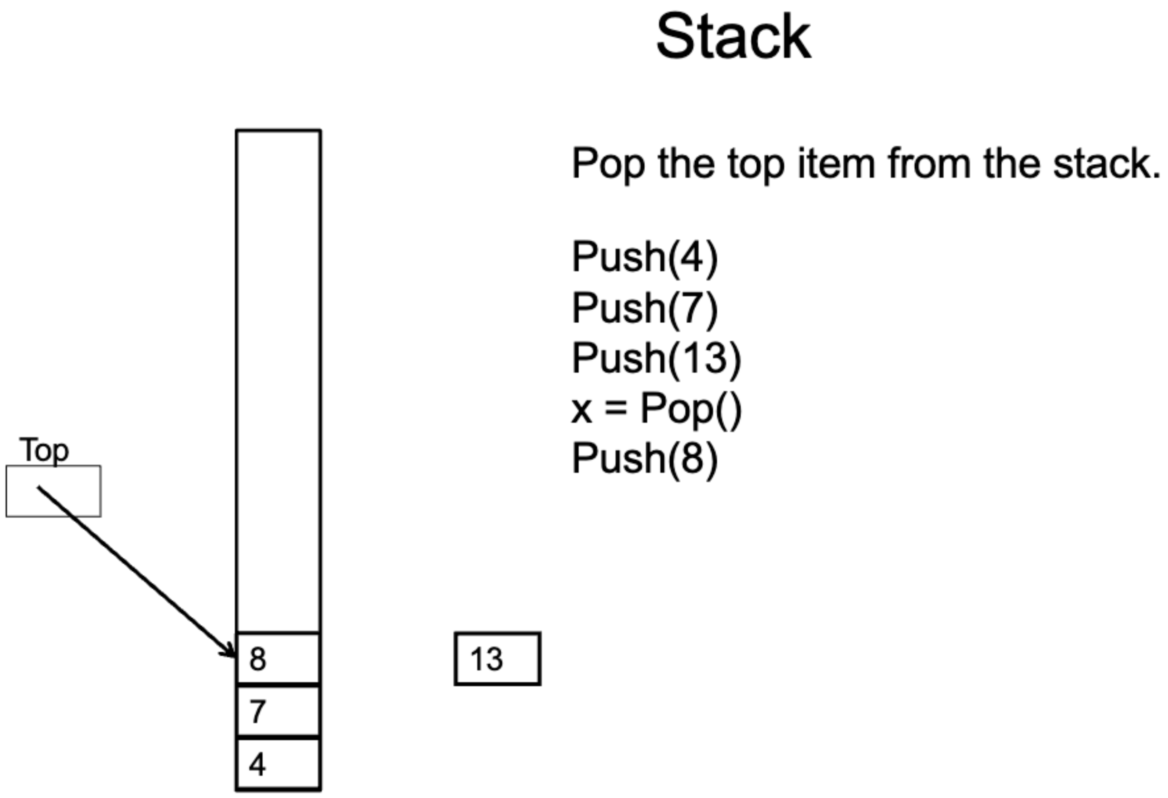
```
if (a->length * 4 <= a->capacity){
  a->capacity /= 2;
  a->array = realloc(a->array, a->capacity * sizeof(int));
}
return true;
```

#ECE

7 Stack

Definition: A data structure for storing a collection of data items, where items can be added to and removed from the collection, but *only the last item added to the stack can be accessed or removed.* This is a last-in, first-out (LIFO) data structure

Common Functions: `push(item)` puts an item on the top of the stack `item=pop()` retrieves the top item from the stack `peek()` retrieves the top item of the stack without removing it



LIFO is important for use cases such as Reversing a word, web browsers to store addresses of recently visited sites, and undo functions in applications

Implementation requirements

Top variable always indicates to the top element of the stack - expect when the stack is empty - special value to indicate the stack is empty

Some mechanism to indicate errors: - attempt to pop an empty stack - out of memory when pushing an element

Example:

`typedef struct { ... } ArrayStk` defines a new struct type for a stack with an alias, holds an index of the current top element in the stack `top`, and sets an array `data` of size 100 to store the stacks elements

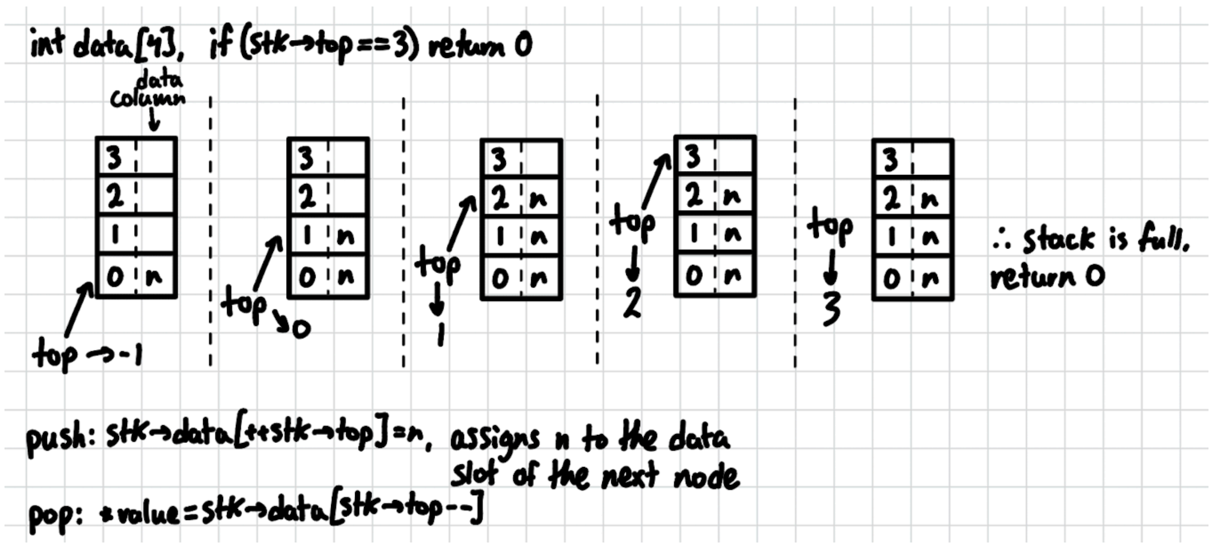
```c
typedef struct {
  int top;               // index of current top, -1 when empty
  int data[4];          // storage
} ArrayStk;
```

This function `push` tries to insert a value `n` onto the stack, and returns 1 if successful and 0 if not. The function also pre-increments `stk->top` (moves it up one position) and stores the new value `n` in that position in the `data` array.

```c
int push(ArrayStk *stk, int n){
  if (stk->top == 3) return 0;         // full
  stk->data[++stk->top] = n;           // push
  return 1;
}
```

The `pop` function checks if full, stores it in that variable pointed to by `value`, then post-decrements `stk->top` (moves it down one position), effectively removing it from the stack

```c
int pop(ArrayStk *stk, int *value){
  if (stk->top == -1) return 0;        // empty
  *value = stk->data[stk->top--];      // pop
  return 1;
}
```


int data[4], if (stk->top==3) return 0
data column
top -> -1
top -> 0
top 1
top 2
top 3
∴ stack is full, return 0
push: stk->data[++stk->top]=n, assigns n to the data slot of the next node
pop: *value=stk->data[stk->top--]

7.1 Modifying the top element

Setup:

```c
ArrayStk stk = { .top = -1 };    // empty stack
int i, value, *ptr_top;          // local vars
```

Pushes 20, 21, 22, 23, 24 onto the stack. Then pops them all off, printing in *reverse order* because of LIFO

```c
for (i=0; i<5; i++) { push(&stk, 20+i); }
while (pop(&stk, &value)) {
  printf("%d\n", value);
}
```

This pushes 100, 101, 102, 103, 104

```c
for (i=0; i<5; i++) { push(&stk, 100+i); }
```

Pops the top into `value`, calls `tos` to return a pointer to the new top element after the pop, then adds 50 to that element in place

pops 104 153 152 151 150

```c
while (pop(&stk, &value)) {
  printf(" %d", value);
  if (tos(stk, &ptr_top))
    *ptr_top = *ptr_top + 50;
}
```

where `tos` is:

```c
int tos(ArrayStk stk)(int **ptop) {
    if (stk.top == -1) return 0;
    *ptop = &stk.data[stk.top];
    return 1;
}
```

7.2 Implementing Stack using Linked List

Building the stack data structure on top of a linked list instead of using an array:

This defines a struct with an alias, and a pointer to the next node in the stack (meaning the node below the top). This allows us to chain nodes like a linked list. We also have `nodval`, which is the actual data being stored in the stack

```c
typedef struct _stk_node {
  struct _stk_node *next;    // link to previous top
  int nodval;                // payload
} StkNode;
```

- Function takes a pointer to the stack head pointer to modify the pointer outside the function
- Allocate memory for a new node on the heap, and stores the new value inside the node

- Links the new node to the current stack and updates the head pointer so the new node is now the stacks top.

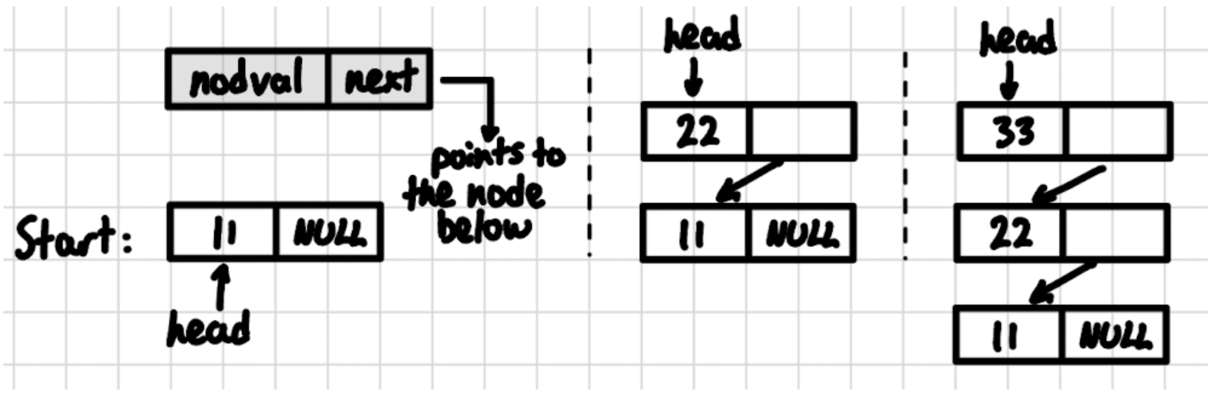
```c
void push(StkNode **stkHead, int n){
  StkNode *p = malloc(sizeof *p);  // allocate new node
  p->nodval = n;                   // set payload
  p->next = *stkHead;              // chain to old top
  *stkHead = p;                    // new node becomes top
}
```

This pop function removes the top node, and returns 0 if the stack is empty (checks for underflow). It firstly saves the current top node pointer, moves the stack head down to the next node, and releases memory for the removed node.

```c
int pop(StkNode **stkHead, int *out){ // (slide shows int return;
 ↪  safer to output)
  if (*stkHead == NULL) return 0;     // underflow guard
  StkNode *p = *stkHead;              // old top
  *out = p->nodval;                   // capture value
  *stkHead = p->next;                 // drop node from stack
  free(p);                            // free storage
  return 1;                           // success
}
```

Top-of-stack returns the address of `nodval` to the head node, else return NULL

```c
int *tos(StkNode *stkHead){
  return stkHead ? &stkHead->nodval : NULL;  // pointer to top value
   ↪  or NULL
}
```



#ECE

8 Queue and Deque

- A linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO)
- Operations: `enqueue` adds an element to the end, `dequeue` removes an element from the front, `isEmpty`
- We need to track the Front (head) and End (tail), thus we need a pointer to the end and front

8.1 Implement Queue with Linked List

Enqueue inserts a node at the rear (end) of the queue

```
void enqueue (int n) {
```

Allocate memory for a new node, assign value and next pointer to NULL

```
struct item *pnew = malloc(sizeof(struct item));
pnew->value = n;
pnew->next = NULL;
```

Nudge n to `pnew`

```
if (end != NULL){
end->next = pnew;
}
if (front == NULL){ // if front is NULL, the list is empty and set
↪  front to new
front = pnew;
}
count++;
```

Dequeue removes an element from the front (head) of the queue

```
bool dequeue (int *n){
```

Set new node `temp` to front Give the pointer value `*n` to front

```
struct item *temp = front;
*n = (front -> value);
```

Move `front` to the next node

```
front = front-> next;
```

Free memory of old front (temp)

```
free(temp);
count--;
return true;
```

8.2 Implement Queue with Array

Common way to implement queue with array is with a circular model. This fixes the idea of a fixed-size array

```
# initialize
// define max size
// define queue array
// define count, front, end = 0
// define isEmpty and isFull functions
```

```
bool enqueue(int n){
// if full, return false
// add to end of queue
end = (end+1) % MAX_SIZE;
count++;
return true;
}
```

```
bool dequeue(int *n){
// if empty, return false
// remove from front of queue
front = (front + 1) % MAX_SIZE;
count--;
return true;
}
```

8.3 Dequeues

- a double-ended queue, can be done with either an array or double linked list
- Operations: `enqueueHead`, `enqueueTail`, `dequeueHead`, `dequeueTail`, `getHead`, and `getTail`

EnqueueHead inserts a new node before the head

```
void enqueueHead (int n) {
struct item *pnew = malloc(sizeof(struct item)); // allocate memory
 ↪  for node

pnew -> value = n;
pnew -> next = NULL;
pnew -> prev = NULL; // assign value and pointers

// update head node
if (head != NULL){
    head -> prev = pnew;
    pnew -> next = head;
}
head = pnew;

// if queue is empty, set tail
if (tail == NULL){
    tail = pnew;
}
count++;
}
```

EnqueueTail inserts a new node after the tail

```
void enqueueTail (int n){
struct item *pnew = malloc(sizeof(struct item)); // allocate memory
 ↪  for node

pnew -> value = n;
pnew -> next = NULL;
pnew -> prev = NULL; // assign value and pointers

// update tail node
if (tail != NULL){
pnew -> prev = tail;
tail -> next = pnew;
}
tail = pnew;

// if queue is empty, set front node
if (head == NULL){
head = pnew;
}
count++:
}
```

DequeueHead removes a node from the head

```
bool dequeueHead(int *n){
struct item *temp = head; // create temp pointer to head

if (count == 0) return false; // queue is empty, can't remove
 ↪  anything

*n = head -> value; // store head value to n

if (head -> next){
head -> next -> prev = NULL; // detach old head
head = head->next; // move head to next node
}

if (head==NULL) tail = NULL; // if head is NULL, queue is empty

free(temp); // deallocate old head node
count--;
return true;
}
```

DequeueTail removes a node from the tail

```
bool dequeueTail (int *n){
struct item *temp = tail; // create temp pointer to head

if (count == 0) return false; // queue is empty, can't remove
 ↪  anything

*n = tail -> value; // store head value to n

if (tail -> next){
tail -> next -> prev = NULL; // detach old head
tail = tail->next; // move head to next node
}

if (tail==NULL) head = NULL; // if head is NULL, queue is empty

free(temp); // deallocate old head node
count--;
return true;
}
```

#ECE

9 Recursion

Sometimes a problem can be solved by first solving a smaller version of the same problem. Recursion means to define something in terms of itself

When the problem is small enough, then it can be solved directly, called the *base case*

Example:

Base Case: $1! = 1$

```
if (n==1){
temp=1;
}
```

Recursive Case $n! = n * (n-1)!$

Iterative vs. Recursive:

Iterative

```
def fact(n){
    result = 1;
    for i in range(1,n+1):
        result *= i;
    return result;
}
print(fact(i))
```

Recursive

```
def fact(n){
    if n==0 or n==1: // Base Case
        return 1;
    else:
        return n * fact(n-1); // Recursive Call
}
print(fact(n))
```

9.1 Linked List - Recursive Insert Sorted

newNode function

```
node *newNode(int value, node *next){
    node *tmp = malloc(sizeof(node));
    tmp -> value = value;
    tmp -> next = next;
    return tmp;
}
```

Insert function

```
int insert(struct node **list, int value){

    // if list empty, or the current node's value is greater than the
    ↪  value to insert, create new node
    if (*list = NULL || (*list)-> value > value){
        *list = newNode(value, *list);
        return 1;
    }

    // if new value is smaller than current, insert it before the
    ↪  current node
    } else if ((*list)->value == value) {
        return 0;

    // otherwise, move one step deeper and try to insert (recursive
    ↪  step)
    else {
        return insert(&((*list)->next), value);
    }
    return 1;
}

int main(){
    struct node *head = NULL;
    insert(&head, 27);  // head: 27
    insert(&head, 92);  // head: 27 -> 92
    insert(&head, 12);  // head: 12 -> 27 -> 92
    insert(&head, 14);  // head: 12 -> 14 -> 27 -> 92
}
```

#ECE

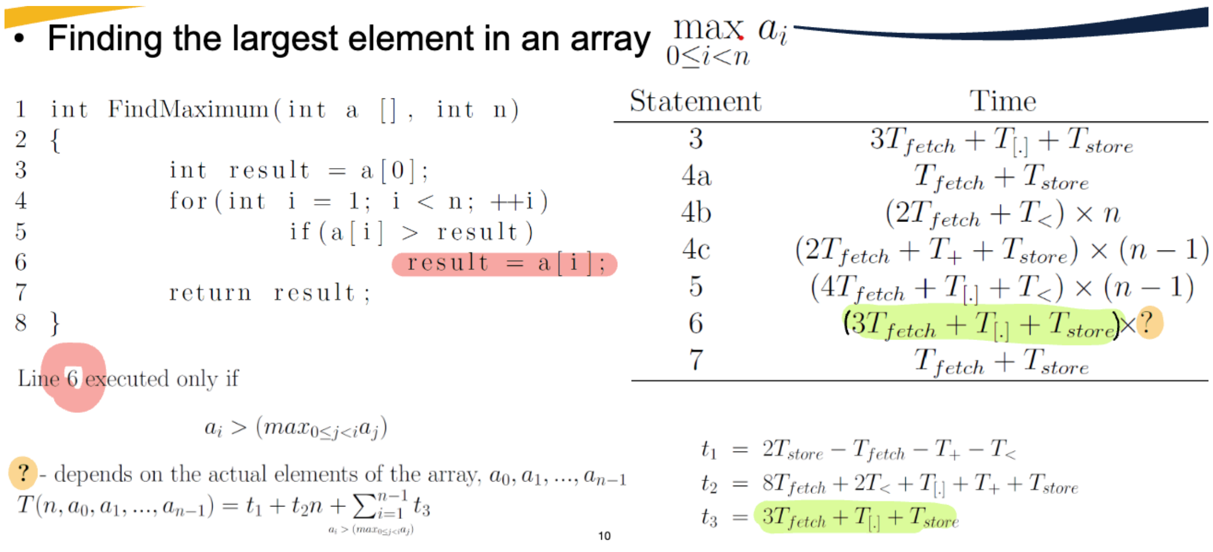
10 Algorithm Analysis

Algorithm analysis is about measuring how much computing resources (like time or memory) an operation or algorithm uses

10.1 Axioms

1. Fetching or storing an integer from memory takes a constant time. $y = x + 1$ takes $T_{fetch} + T_{store}$
2. Basic operations (add, subtract, multiply, divide, compare) on integers all take constant time. $y = y + 1$ takes $2T_{fetch} + T_{op} + T_{store}$
3. Function call and return times are constant (T_{call}, T_{return}), passing an integer argument is like fetching it. $y = f(x)$ takes $T_{fetch} + T_{call} + T_{f(x)} + T_{store}$
4. Array subscripting address calculation is constant T, not including time to compute i or fetch/store the element. $y = a[i]$ takes $3T_{fetch} + T_{store} + T$
5. Allocating memory is constant time T_{new}, not including initialization

10.2 Examples



- Finding the largest element in an array $\max\limits_{0 \le i < n} a_i$

```
1  int FindMaximum(int a [], int n)
2  {
3          int result = a[0];
4          for(int i = 1; i < n; ++i)
5                  if(a[i] > result)
6                          result = a[i];
7          return result;
8  }
```

Statement	Time
3	$3T_{fetch} + T_{[.]} + T_{store}$
4a	$T_{fetch} + T_{store}$
4b	$(2T_{fetch} + T_{<}) \times n$
4c	$(2T_{fetch} + T_{+} + T_{store}) \times (n-1)$
5	$(4T_{fetch} + T_{[.]} + T_{<}) \times (n-1)$
6	$(3T_{fetch} + T_{[.]} + T_{store}) \times ?$
7	$T_{fetch} + T_{store}$

Line 6 executed only if

$$a_i > (max_{0 \le j < i} a_j)$$

? - depends on the actual elements of the array, $a_0, a_1, ..., a_{n-1}$
$T(n, a_0, a_1, ..., a_{n-1}) = t_1 + t_2 n + \sum_{i=1}^{n-1} t_3$ $\scriptstyle a_i > (max_{0 \le j < i} a_j)$

$$\begin{aligned} t_1 &= 2T_{store} - T_{fetch} - T_{+} - T_{<} \\ t_2 &= 8T_{fetch} + 2T_{<} + T_{[.]} + T_{+} + T_{store} \\ t_3 &= 3T_{fetch} + T_{[.]} + T_{store} \end{aligned}$$

10

$T_{average}(n) = t_1 + t_2 n + t_3 \Sigma_{i+1}^{n-1} \frac{1}{i+1}$

The probability that a_i is the largest of the $i + 1$ values, which is $\frac{1}{i+1}$ from $p_i = P[a_i > (max_{0 \le j < i} a_j)]$

$T_{worsecase}(n) = (t_1 - t_3) + (t_2 + t_3) \times n$ $T_{bestcase}(n) = t_1 + t_2 n + \Sigma_{i=1}^{n-1} p_i t_3$ when $p_i = 0$, thus $= t_1 + t_2 n$

Instead of detailing each parameter, we let T be the *clock cycle* of a machine, where $T_{fetch} = kt$

Assume that all timing parameters expressed in units of clock cycles. Thus $T = 1$. k is assumed to be the same for all parameters, thus $k = 1$

#ECE

11 Big Oh Notation

11.1 Time Functions

The time function $T(n)$ describes how the running time of an algorithm grows as a function of the input size n

$f(n) = O(g(n))$ means for large enough n, $f(n)$ is at most a constant times $g(n)$

Example: Given $8n + 128$, show that $f(n) = O(n^2)$, find constants $n_0 > 0$ and $c > 0$ such that $\forall n \geq n_0, f(n) \leq cn^2$

If $c = 1$, then $f(n) \leq cn^2 \Rightarrow 8n + 128 \leq n^2 \Rightarrow 0 \leq (n - 16)(n + 8)$

Since $(n + 8) > 0$ for all values $n \geq 0$, then $(n_0 - 16) \geq 0, i.e. n_0 = 16$

Possible solution: for $c = 1, n_0 = 16, f(n) \leq cn^2$ for all integers $n \geq n_0$, hence $f(n) =)(n^2)$

11.2 Properties of $O()$

If two functions have the same $O(g(n))$, the functions are not necessarily equal, they just share the same upper bound on their growth

When adding functions, we take $f_1(n) + f_2(n) = O(max(g_1(n), g_2(n)))$

If we multiply two functions, we get $O(n^2) + O(n^3) = O(n^5)$

If we have a polynomial time or space functions, we take the term with the highest order without the constant, e.g. $f(n) = O(n^m)$

Big O: upper bound (worst-case growth) Big Ω: lower bound (best-case growth) Big Θ: tight bound (exact growth)

11.3 Sum Example

This algorithm sums elements in an array, let $n = 5$

```
int sum(a,n){
    s=0;
    for(int i = 0; i<n; i++){
        s=s+a[i];
    }
    return s;
}
```

frequency count method:

```

// 1
// for (1, n+1, n), use the greatest count (n+1)
// n
// 1

// time function f(n) = 2n+3
// O(n)
```

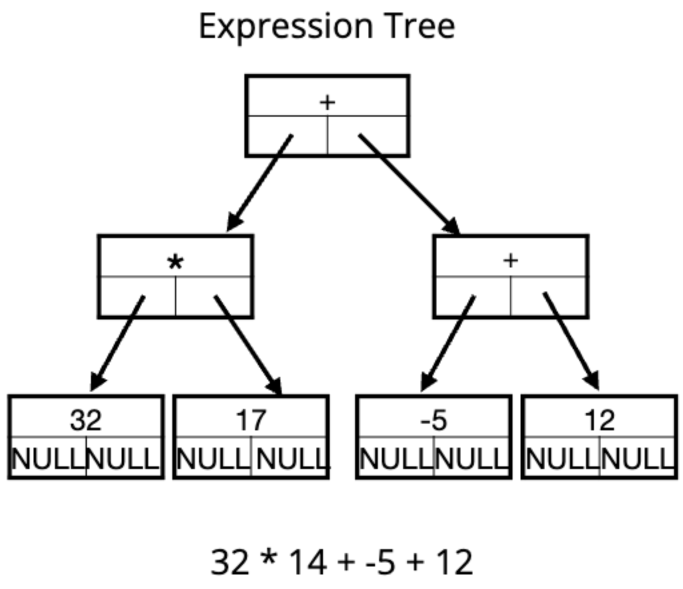
#ECE

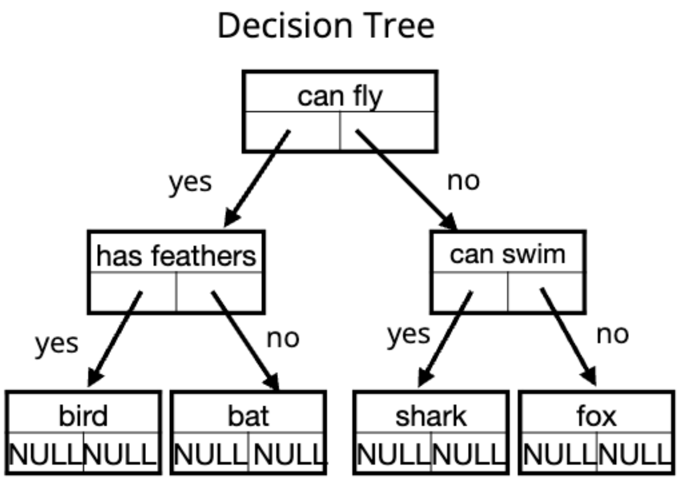
12 Trees

Trees are one of the most fundamental concepts in programming.

Trees are a linked data structure, with nodes and pointers, are are used to represent hierarchical data, organize info for searching, make decisions, etc.

Examples include family trees, search based on car brand -> model -> year -> colour

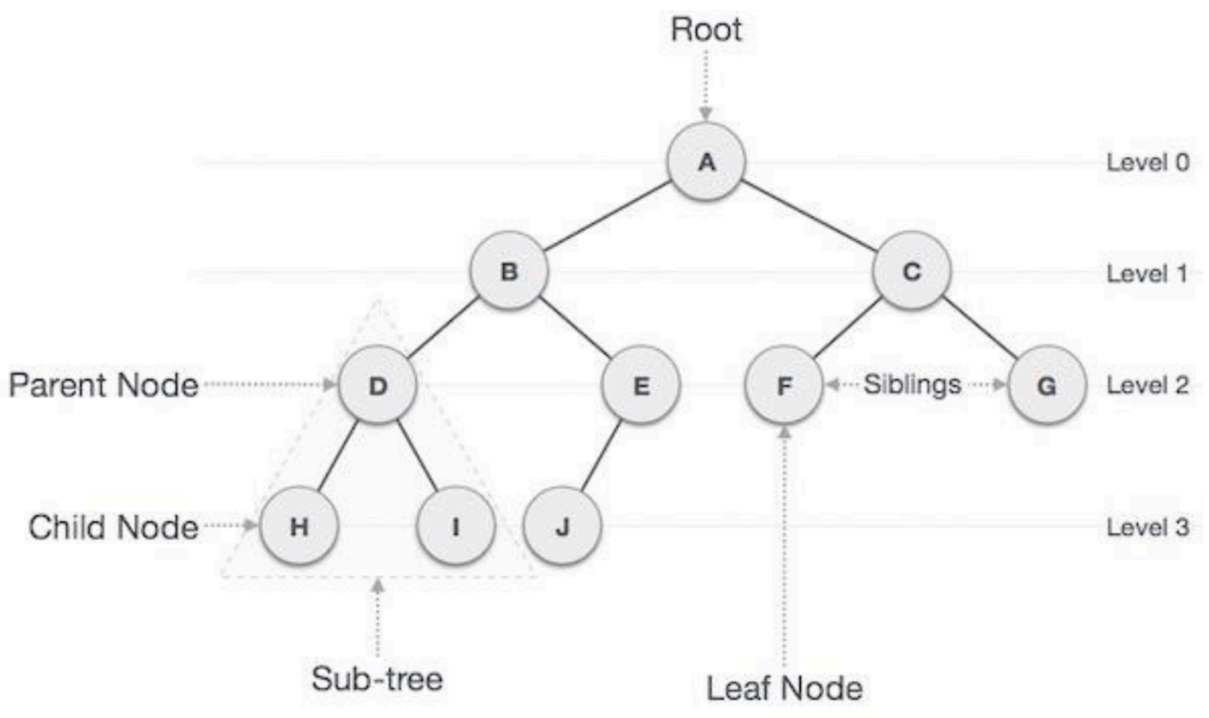




12.1 Terminology

Trees contain nodes, where every node may have zero or more children. Children nodes are accessed via their parent node, pointing downwards

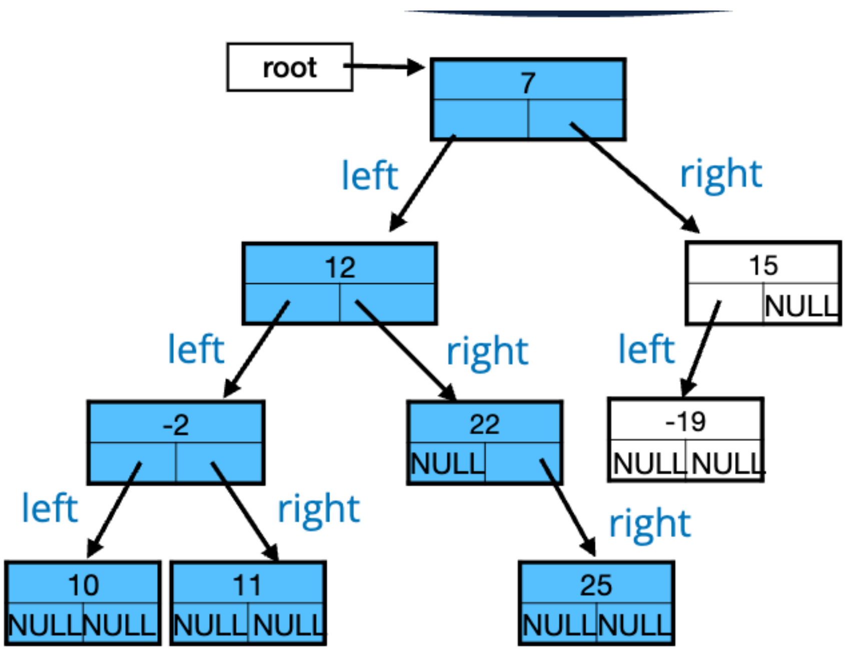
A **binary** tree has 0, 1 or 2 children. Every tree has a **root**, which is the top node.



12.2 Implementation

```c
typedef struct node {
    int data;
    struct node * left;
    struct node * right;
} node;

new* newNode(int data) {
    node *nptr = (node*) malloc (sizeof(struct node));
    nptr->data = data;
    nptr->left = NULL;
    nptr->right = NULL;
    return nptr;
}
```



```
page 19
```

adding a node

```
page 20
```

12.3 More terminology

Edges are the relationship between a parent and a child in a tree, drawn as a line and implemented as a pointer. e.g. $\{AB, AC, BH, etc.\}$

A **path** starts from one node and ends at another node, strictly downwards and identifies all nodes and edges along the way. e.g. Path $= ABH$. There is exactly one path from root to any given node.

The **height** of a node is the number of edges in the longest downward path from that node and a leaf node (root down)

The **depth** of a node is the number of edges from the root to the node, essentially the opposite from the height (leaf up)

The **level** of a node is the depth of the node $+1$

12.4 Types of Trees

A **subtree** is a given node in the tree and *all* of the nodes below it. Leaf nodes have empty subtrees.

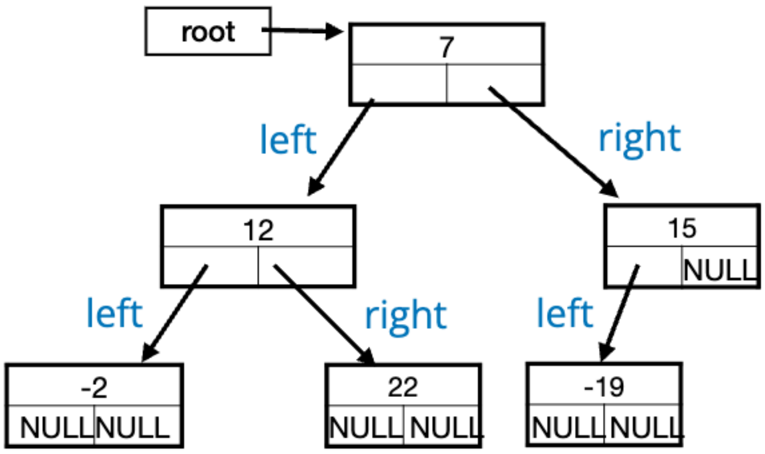
Binary trees are trees with at most two children for any node. A binary tree is **full** if all nodes have exactly zero (i.e. leaf) or two. A binary tree is **complete** if all levels except the last are filled (all nodes in the last level must be as far left as possible).

12.5 Binary Tree Operations

We can traverse all the nodes, search, add, add at a specific location, delete, delete the entire tree

We can visit all the nodes in a tree starting at the root, but in four common orders: - pre-order - in-order - post-order - level-order

It is important to note that trees, by nature, are recursive.



Example tree:

12.5.1 Pre-order Traversal

Print the root first, then left, then right

1. process the current node
2. traverse the current nodes left subtree
3. traverse the current nodes right subtree
4. process the current node

```
preorderPrint(BinaryTreeNode *cur){
    if (cur == NULL) return; // check
    else {
        printf("%d \n", cur->data);
        preorderPrint(cur->left); // recursion for left
        preorderPrint(cur->right); // recursion for right
    }
}

preorderPrint(root);
```

Output: 7 12 -2 22 15 -19

12.5.2 In-order traversal

Print left child first, then root, then right child

1. traverse the current nodes left subtree
2. process the current node
3. traverse the current nodes right subtree

```
inorderPrint(BinaryTreeNode *cur){
    if (cur == NULL) return; // check
    else {
        inorderPrint(cur->left); // recursion for left
        printf("%d \n", cur->data);
        inorderPrint(cur->right); // recursion for right
    }
}

inorderPrint(root);
```

Output: -2 12 22 7 -19 15

12.5.3 Post-order traversal

Print value of root last

1. traverse the current nodes left subtree
2. traverse the current nodes right subtree
3. process the current node

```
postorderPrint(BinaryTreeNode *cur){
    if (cur == NULL) return; // check
    else {
        postorderPrint(cur->left); // recursion for left
        inorderPrint(cur->right); // recursion for right
        printf("%d \n", cur->data);
    }
}

postorderPrint(root);
```

Output: -2 22 12 -19 15 7

12.6 Level-order traversal

start at the root, traverse each level from left to right difficult to do recursively, use a queue instead

1. add the root node to a queue
2. while the queue is not empty
    1. remove the first element form the queue
    2. process the node
    3. add any children from left to right to the queue

```
page 96
```