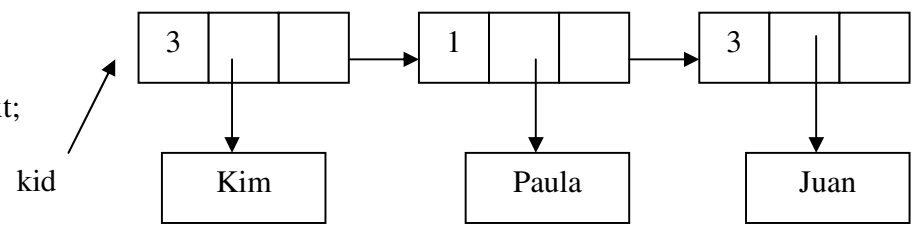
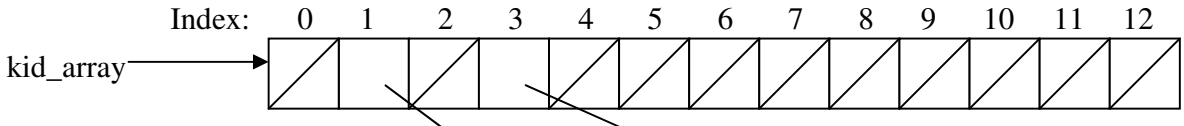
Question 1: You must be kidding! (groan) (15 pts, 40 min)

We have a simple linked list that consists of kids' names (a standard C string) and the grade they are in – an integer between 0 (Kindergarten) and 12. The structure appears as follows, with an example:

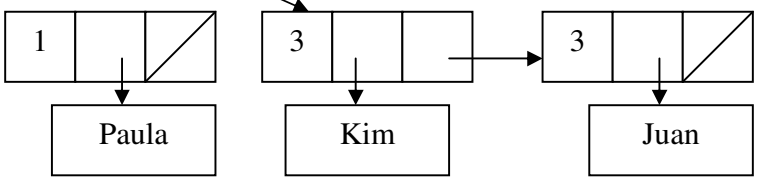
```
typdef struct kid_node {
        int grade;
        char *name;
        struct kid_node *next;
} kid_t;
```



For "administrative reasons", we'd like to categorize our kids by **grade.**
We **copy** the kids' information into an array of linked lists indexed by the grade.



Note that changing ANY of the data
In these structures here should NOT
Affect the original list above.

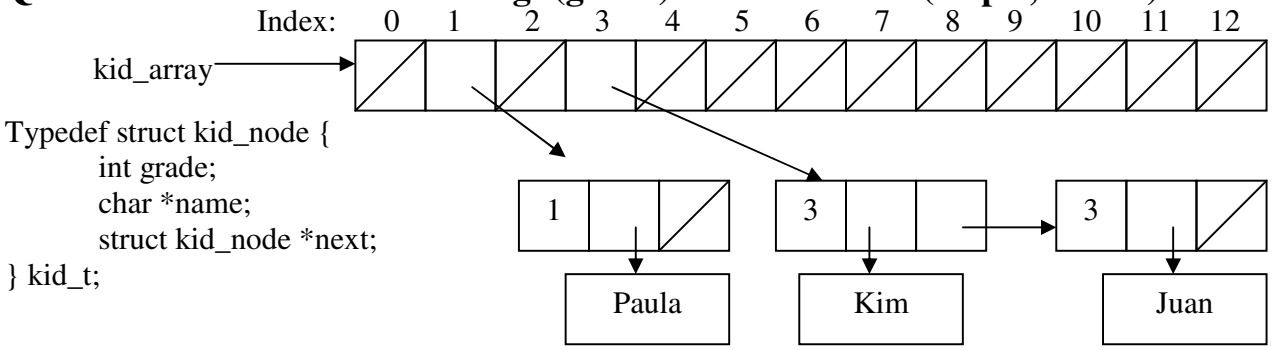


Fill in the blanks in the below code:

a) The **create_kid_array** function will return a pointer to the new array. Remember, the range of grades is 0-12, inclusive, and the original list MUST remain unchanged.

```
#define MAXGRADE 12
kid_t   **create_kid_array (kid_t *kid) {
        int i;              /* in case you need an int */
        kid_t *temp;    /* or kid_t ptr somewhere */
        kid_t **kid_array = (kid_t **) malloc (kid_t** malloc((MAXGRADE+1)*sizeof(kid_t*));
        if (kid_array == NULL) return NULL;
        /* Additional initializing – add some code below */
        for( i = 0; i <= MAXGRADE; i++ ) {  kid_array[i] = NULL; }
        while( kid != NULL ) {
          temp = (kid_t*) malloc(sizeof(kid_t));
          temp->name = (char*) malloc(strlen(kid->name)+1)*sizeof(char));
          strcpy( temp->name, kid->name );
          temp->grade = kid->grade;
          temp->next = kid_array[grade];      kid_array[grade] = temp;
          kid = kid->next;
        }
        ret
}
```

Question 1: You must be kidding! (groan) …continued… (15 pts, 40min)




```
Typedef struct kid_node {
        int grade;
        char *name;
        struct kid_node *next;
} kid_t;
```

b) For every Yin, there is a Yang. Now that we have a function for **creating** kid arrays, we must create a function that **frees** all memory associated with the structure. Fill in the following functions. **free_kid_array** calls the *recursive* function **free_kid_list** which frees a single kid list.

```
void free_kid_array(kit_t *kid_array[]){
    int i;
    for (i = 0; i<= MAXGRADE; i++){
            free_kid_list(kid_array[i]);
    }
    /* Clean up if necessary */
    free( kid_array );
}

void free_kid_list(kid_t *kid) {
    /* Don't need anything here... */          /* Declare temp variables */

    if (kid == NULL)
            return;
    free_kid_list( kid->next );
    free( kid->name );
    free( kid );
}
```

Question 2: C (16 Points – 30 minutes)

We've written **matchSubStr** below in C. Some of the lines are buggy and some are perfectly fine. Circle BUGGY or OK for each labeled C statement and if buggy, why it is and provide the fix. A line may be buggy for multiple reasons, so be sure you're descriptive.

Use the comments near each statement as a guide for what the line SHOULD do. If the code is buggy and you have a more clever/intuitive way of doing the same thing, feel free to do it your way. Note: You can assume only valid input will be provided (two non-empty, null-terminated strings).

```
/* This function tries to find a substring (sub) within another (string).
 * If matchSubStr() finds the substring, it returns the index of the start
 * of the substring.  If there is more than one match, it returns the first.
 * This is the scheme-equivalent of an equal? match (not eq? match) */
int matchSubStr(char sub[], char string[]) {

/* Holds the location we're checking (and will return if a match). */
A:  int loc;          || BUGGY If buggy, why?
                      || OK    If buggy, fix:

    /* These variables are pointers to the chars in sub/string */
B:  char *c1, c2;     || BUGGY If buggy, why?
                      || OK    If buggy, fix:

    /* We want to iterate through the string looking for a match, so we start at
     * loc=0 (beginning) and keep going as long as we have characters remaining */
C:  for(loc=0; strlen(string[loc]); loc++) { || BUGGY If buggy, why?
                                             || OK    If buggy, fix:

        /* We step through the substring using c1 and c2 to reference the
         * letters in sub and string.  We stop when we have either exhausted
         * all the characters in sub (and thus found a match) or when we
         * encounter two characters that are not equivalent. */

D:      for(c1 = sub, c2 = string&loc;  || BUGGY If buggy, why?
                                        || OK    If buggy, fix:

E:          ;                           || BUGGY If buggy, why?
                                        || OK    If buggy, fix:

F:          c1++, c2++) {               || BUGGY If buggy, why?
                                        || OK    If buggy, fix:

            /* If we didn't find a match, we break out */
G:          if(c1 != c2) {  || BUGGY If buggy, why?
                            || OK    If buggy, fix:
                break;
            }
        }

        /* We return the location if we found a match */
H:      if(1) {             || BUGGY If buggy, why?
                            || OK    If buggy, fix:
            return loc;
        }
    }
    /* Return -1 if we didn't find a match */
    return -1;
}
```

Looks good!

Buggy. c2 needs to be a pointer!
`char *c1, *c2;`

Buggy. string[loc] is not a string! Should be:
`strlen( (char*) string+loc )`

Buggy. string&loc is meaningless.
`c2 = string+loc`

Buggy! We want to break at the end of either string.
`*c1 != '\0' && *c2 != '\0';`

Looks good!

Buggy. c1 and c2 are pointers! Should be:
`if( *c1 != *c2 ) {`

Buggy! We only have found a substring if we're at the end of sub! Should be:
`if( *c1 == '\0' )`

Question 3: Numerical Representation (10 points – 20 min.)

Considering *8-bit integers*, answer the following questions for each column. The bits are numbered as: 7 6 5 4 3 2 1 0. Each box might be a different integer. You *must* show scratch work to receive credit.

		Given that bits 3-0 are 1111		**Given that bits 7-4 are 1001**	
If the # were interpreted as a two's complement signed integer, would it be negative (-), positive (+) or impossible to tell? (circle one)		- CAN'T-TELL +		- CAN'T-TELL +	
If the # were interpreted as a [each of the values on the right], what is the most negative (closest to -∞) value possible? (for each answer, show your work and write the decimal and hexadecimal value immediately below)	**Sign-magnitude**	*Scratch space* *Decimal Value* -127	*Hexadecimal Value* 0xFF	*Scratch space* *Decimal Value* -31	*Hexadecimal Value* 0x9F
	Unsigned	*Scratch space* *Decimal Value* -15	*Hexadecimal Value* 0x0F	*Scratch space* *Decimal Value* -144	*Hexadecimal Value* 0x90
	Two's complement signed	*Scratch space* *Decimal Value* -113	*Hexadecimal Value* 0x8F	*Scratch space* *Decimal Value* -112	*Hexadecimal Value* 0x90
	One's complement	*Scratch space* *Decimal Value* -112	*Hexadecimal Value* 0x8F	*Scratch space* *Decimal Value* -111	*Hexadecimal Value* 0x90