Final Exam will be in class on **Wed, Dec 11, 8:00-10:00am**. The exam will be closed-book, closed-notes, but you will be allowed 3 *sheets of notes*, front and back (handwritten or typeset, your choice). Please plan to bring your university ID with you during the exam.

**Disclaimer:** This just reflects the material since the second midterm. These practice problems have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** necessarily reflect the actual coverage, difficulty, or length of the midterm exam.

**Problem 1.** Since the exam is comprehensive, please look over all the old homework assignments, exams, and practice problems.

**Problem 2.** Recall the buddy system of allocating blocks of memory (see Fig. 1). Throughout this problem you may use the following standard bit-wise operators:

<table>
<thead>
<tr>
<th>Bit-wise Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bit-wise “and”</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise “exclusive-or”</td>
</tr>
<tr>
<td>~</td>
<td>bit-wise “complement”</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift (filling with zeros)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift (filling with zeros)</td>
</tr>
</tbody>
</table>

You may also assume that you have access to a function `bitMask(k)`, which returns a binary number whose `k` lowest-order bits are all 1’s. For example `bitMask(3) = 111_2 = 7`.

![Figure 1: Buddy relatives.](image)

Present a short (one-line) expression for each of the following functions in terms of the above bit-wise functions:

(i) `boolean isValid(int k, int x)`: True if and only if `x ≥ 0` a valid starting address for a buddy block at level `k ≥ 0`.

(ii) `int sibling(int k, int x)`: Given a valid buddy block of level `k ≥ 0` starting at address `x`, returns the starting address of its *sibling*. 
(iii) **int parent(int k, int x):** Given a valid buddy block of level \( k \geq 0 \) starting at address \( x \), returns the starting address of its parent at level \( k + 1 \).

(iv) **int left(int k, int x):** Given a valid buddy block of level \( k \geq 1 \) starting at address \( x \), returns the starting address of its left child at level \( k - 1 \).

(v) **int right(int k, int x):** Given a valid buddy block of level \( k \geq 1 \) starting at address \( x \), returns the starting address of its right child at level \( k - 1 \).

**Problem 3.** Suppose you have a large span of memory, which starts at some address \( \text{start} \) and ends at address \( \text{end} - 1 \) (see Fig. 2). (The variables \( \text{start} \) and \( \text{end} \) are generic pointers of type \( \text{void*} \).) As the dynamic memory allocation method of Lecture 15, this span is subdivided into blocks. The block starting at address \( p \) is associated with the following information:

- \( p\text{.inUse} \) is 1 if this block is in-use (allocated) and 0 otherwise (available)
- \( p\text{.prevInUse} \) is 1 if the block immediately preceeding this block in memory is in-use. (It should be 1 for the first block.)
- \( p\text{.size} \) is the number of words in this block (including all header fields)
- \( p\text{.size2} \) each available block has a copy of the size stored in its last word, which is located at address \( p + p\text{.size} - 1 \).

(For this problem, we will ignore the available-list pointers \( p\text{.prev} \) and \( p\text{.next} \).)

In class, we said that in real memory-allocation systems, blocks cannot be moved, because they may contain pointers. Suppose, however, that the blocks are movable. Present pseudo-code for a function that compacts memory by copying all the allocated blocks to a single contiguous span of blocks at the start of the memory span (see Fig. 2). Your function \( \text{compress}(\text{void* start, void* end}) \) should return a pointer to the head of the available block at the end. Following these blocks is a single available block that covers the rest of the memory’s span.

To help copy blocks of memory around, you may assume that you have access to a function \( \text{void* memcpy(\text{void* dest, void* source, int num})} \), which copies \( \text{num} \) words of memory from the address \( \text{source} \) to the address \( \text{dest} \).