Deallocation Models:

Explicit: (C++, C++)
- programmer deletes
- may result in leaks, if not careful
Implicit: (Java, Python)
- runtime system deletes
- Garbage collection
- Slower runtime
- Better memory compaction

Explicit Allocation/Deallocation
- Heap memory is split into blocks whenever requests made
- Available blocks:
  - stored in available block list
  - merged when contiguous

Runtime System Mem. Mgr.
- Stack - local vars, recursion
- Heap - for "new" objects
  - Don't confuse with heap data structure/heapsort

What happens when you do
- new (Java)
- malloc/free (C)
- new/delete (C++)?

Block Structure:
Allocated:
- prevInUse: 1 if prev. contig. block is allocated
- prev/next: links in avail. list
- size, size2: total block size (includes headers)

Available:
- prevInUse

Memory Management I

Fragmentation:
- Results from repeated allocation/deallocation (Swiss-cheese effect)

Guide:
- First-fit: Take first block from avail. list that is large enough
- Best fit: Find closest fit from avail list

How to Select from available blocks?
- External: Caused by pattern of alloc/dealloc
- Internal: Induced by mem. manage. policies (not user)
- Surprise: First-fit is usually better - faster & avoids small fragments
Example: Alloc \( b = 59 \)

Allocation:
- \( \text{malloc}(b) \)
  - Search avail. list for block of size \( b' \geq b+1 \)
  - If \( b' \) close to \( b \): alloc entire block (unlink from avail list)
  - Else: split block

Deallocation:
- If prev\(\Rightarrow\)next contiguous blocks are allocated \( \rightarrow \) add this to avail
- Else: merge with either/both to make max. avail block

Example:

Memory Management

Some C-style pointer notation
- \( \text{void}^* \) - pointer to generic word of memory
- Let \( p \) be of type \( \text{void}^* \):
  - \( p+10 \) - 10 words beyond \( p \)
  - \( *(p+10) \) - contents of this
- Let \( p \) point to head of block:
  - \( p\text{.inUse, p.prevInUse, p.size} \)
    - We omit bit manipulation
  - \( *(p+p\text{.size}-1) \) - references last word in this block

\begin{verbatim}
(void*) alloc (int b) {
    b += 1 \( // \text{add } 1 \text{ for header} \)
    p = \text{search avail. list for block size } \geq b
    if (p == null) Error-Out of mem!
    if (p\text{.size} - b \leq TOO\_SMALL)
        unlink \( p \) from avail. list
        \( q = p \)
    else .... (continued)
}
\end{verbatim}

\begin{verbatim}
else {
    p\text{.size} -= b \( // \text{remove allocation} \)
    \( *(p+p\text{.size}-1) = p\text{.size} \)
    \( q = p + p\text{.size} \) \( // \text{start of new block} \)
    q\text{.size} = b
    q\text{.prevInUse} = 0 \( // \text{new block header} \)
    q\text{.inUse} = 1
    (q + q\text{.size}), prevInUse = 1 \( // \text{update prevInUse for next contig. block} \)
    return \( q+1 \) \( // \text{skip over header} \)
}
\end{verbatim}
Buddy System:
- Block sizes (including headers) are power of 2
- Requests are rounded up (internal fragmentation)
- Block size $2^k$ starts at address that is multiple of $2^k$
- $k$ = level of a block

Structure:

<table>
<thead>
<tr>
<th>Level</th>
<th>Block Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[0, 15]</td>
</tr>
<tr>
<td>3</td>
<td>[0, 7)</td>
</tr>
<tr>
<td>2</td>
<td>[0, 3)</td>
</tr>
<tr>
<td>1</td>
<td>[0, 1)</td>
</tr>
<tr>
<td>0</td>
<td>[0, 0]</td>
</tr>
</tbody>
</table>

In practice: There is a minimum allowed block size.
Buddy system only allows allocations aligning with these blocks.

Coping with External Fragmentation
- Unstructured allocation can result in severe external fragmentation.
- Can we compress? Problem of pointers.
- By adding more structure we can reduce external frag. at cost of internal frag.

Memory Management

Merging:
- When two adjacent blocks are available, we don't always merge them.
  - Must have same size: $2^k$
  - Must be buddies/siblings in the tree structure.

Def: $\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } 2^k \text{ divides } x \\ x - 2^k & \text{otherwise} \end{cases}$

$\equiv \text{buddy}_k(x) = (1 < k) \oplus x$ [Bit manipulation]

Example: $\text{alloc}(2) \rightarrow \text{alloc}(4)$
- $k = \lceil \lg (b+1) \rceil$
- if $\text{avail}[k]$ non-empty - return entry, delete
- else: find $\text{avail}[j] \neq \emptyset$ for $j > k$
  - split this block

Big Picture:
- Avail list is organized by level: $\text{avail}[k]$
- Block header structure same as before except:
  - $\text{prevInUse} \neq \emptyset$ not needed size $2^k$