Data Structures

Mapping complex structures
to linear memory
Computer memory

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>0C</td>
<td>0D</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>
Mapping a binary tree to memory

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
</tr>
</thead>
<tbody>
<tr>
<td>07</td>
<td>Null</td>
<td>3</td>
<td>05</td>
<td>Null</td>
<td>4</td>
</tr>
<tr>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>0E</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Null</td>
<td>9</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0A</td>
<td>2</td>
<td>02</td>
<td></td>
</tr>
</tbody>
</table>

Tree diagram:

```
  5
 / \
2   9
/   /\   /
1 3 11 4
```
What tree is this?

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A</td>
<td>0D</td>
<td>3</td>
<td>NULL</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
</tr>
<tr>
<td>NULL</td>
<td>5</td>
<td>13</td>
<td>02</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>NULL</td>
<td>1</td>
<td>16</td>
<td>07</td>
<td>11</td>
<td>NULL</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>NULL</td>
<td>9</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Ideas

- Using an address to refer to a value by the value’s location in memory:

- Using adjacency to create relationship
“C” syntax

```c
int a, b;
int* aPointer;

a = 6;
aPointer = &a;
b = *aPointer;
```
"C" syntax

```c
struct Node {
    Node* left;
    int number;
    Node* right;
};

Nodes node1, node2, node3;
```
“C” syntax

struct Node {
    Node* left;
    int number;
    Node* right;
}

Node node1, node2, node3;

node1.number = 11;
node1.left = NULL;
node1.right = NULL;
"C" syntax

```c
struct Node {
    Node* left;
    int number;
    Node* right;
}

Node node1, node2, node3;

node2.number = 9;
node2.right = &node1;
node2.left = NULL;
```
Dynamic memory allocation

Q: Do all the nodes have to be allocated in advance?
A: No. They can be dynamically allocated.

Node* nodex;

nodex = malloc (sizeof (Node));

Note: there is no “name” for the allocated space; only a pointer to the beginning of the space.
Using dynamic memory

Node* nodex;

nodex = malloc( sizeof(Node));
(*nodex).number = 4;
(*nodex).left = NULL;
(*nodex).right = NULL;
Dynamically created tree

```
    bTree
     /   \
   /     /
left 5 right
     /   \
   /     /
left 2 right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /   \
   /     /
left NULL right
     /
```
Traversals

In the dynamically created tree, what would you write to assign to “r” the value in the box assuming “bTree” points to the element containing the value “5”?

```c
struct Node {
    Node* left;
    int number;
    Node* right;
};
Node* bTree;
int r;
```
Types of storage

- **Stack (local) storage**
  - Lifetime: only during execution of function/method
  - Allocation/deallocation: automatic
  - Problem: violation of lifetime

- **Heap (global) storage**
  - Lifetime: indefinite
  - Allocation/deallocation: programmed using malloc and free
  - Problem: memory leaks (memory allocated and not deallocated)
### Stack storage

```c
void X() {
    int a = 1;
    int b = 2;
    // T1
    Y(a);
    // T3
    Y(b);
    // T5
}
void Y(int p) {
    int q;
    q = p + 2;
    // T2 (first time through), T4 (second time through)
}
```

<table>
<thead>
<tr>
<th></th>
<th>T1 - X()'s locals have been allocated and given values.</th>
<th>T2 - Y() is called with p=1, and its locals are allocated. X()'s locals continue to be allocated.</th>
<th>T3 - Y() exits and its locals are deallocated. We are left only with X()'s locals.</th>
<th>T4 - Y() is called again with p=2, and its locals are allocated a second time.</th>
<th>T5 - Y() exits and its locals are deallocated. X()'s locals will be deallocated when it exits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X()</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Violation of lifetime

// TAB -- The Ampersand Bug function
// Returns a pointer to an int
int* TAB() {
    int temp;
    return (&temp); // return a pointer to the local int
}

void Victim() {
    int* ptr;
    ptr = TAB();
    *ptr = 42;    // Runtime error! The pointee was local to TAB
Heap storage

allocation: `void* malloc (unsigned long size);`

deallocation: `void free (void* block);`

3 separate heap blocks — each 1024 bytes in size.
Heap storage

```c
void Heap1() {
    int* intptr;
    // Allocates local pointer local variable (but not its pointee)
    // T1
    intptr = malloc(sizeof(int));
    *intptr = 42;
    // T2
    intptr = malloc(sizeof(int));
    *intptr = 42;
    // T3
    free(intptr);
    free(intptr);
}
```
void HeapArray() {
    struct fraction* fracts;
    int i;

    // allocate the array
    fracts = malloc(sizeof(struct fraction) * 100);

    // use it like an array -- in this case set them all to 22/7
    for (i=0; i<99; i++) {
        fracts[i].numerator = 22;
        fracts[i].denominator = 7;
    }

    // Deallocate the whole array
    free(fracts);
}
Automatic storage management

- Memory management problems
  - easy to cause
  - difficult to debug
    - unusual failure modes
    - may be difficult to know what component is responsible for deallocating memory

- Solutions
  - C/C++: tools and libraries for “safe” memory management or debugging assistance
  - Java: automatic storage management (garbage collection)
Garbage collection

- Goal: automatically detect and reclaim allocated but unusable memory

- Basic approaches
  - Mark-and-sweep
  - Copying collectors

- Costs
  - Overhead of garbage collector
  - Acceptable in most cases (esp. in light of advantages)
Mark and sweep

- Basic algorithm
  - Starting from the program variables, mark all memory blocks encountered
  - Sweep through all memory block in the heap and reclaim the unmarked ones
  - Unmark all marked memory blocks
Copying collector

- Basic algorithm
  - Divide memory into two regions
  - Begin allocating from region 1
  - When region 1 is full
    - Copy all usable blocks from region 1 to region 2
    - Interchange roles for region 1 and region 2
Memory hierarchy
Memory hierarchy

- Why does this matter? To algorithm design?

<table>
<thead>
<tr>
<th>Memory</th>
<th>Access Time</th>
<th>Normalized Human</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>1 nsec</td>
<td>1 sec</td>
<td>on desk</td>
</tr>
<tr>
<td>Cache</td>
<td>20 nsec</td>
<td>20 sec</td>
<td>in room</td>
</tr>
<tr>
<td>Main Memory</td>
<td>50 nsec</td>
<td>1 minute</td>
<td>next door</td>
</tr>
<tr>
<td>Disk</td>
<td>10 msec</td>
<td>100 days</td>
<td>off planet</td>
</tr>
</tbody>
</table>