Data Structures

Mapping complex structures
to linear memory

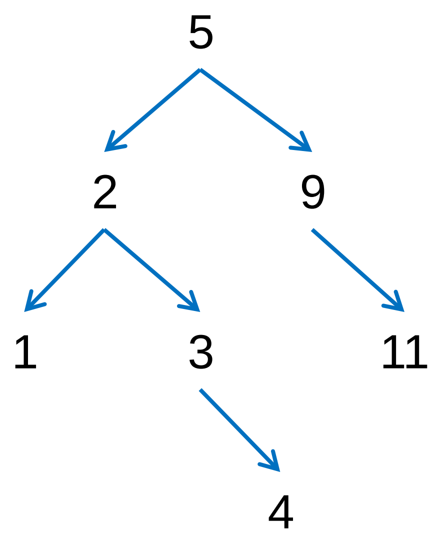
Computer memory

00	01	02	03	04	05
06	07	08	09	0A	0B
0C	0D	0E	0F	10	11
12	13	14	15	16	17

address
value

Mapping a binary tree to memory

00	01	02	03	04	05
07	Null	3	05	Null	4
06	**07**	**08**	**09**	**0A**	**0B**
16	5	0E		1	
0C	**0D**	**0E**	**0F**	**10**	**11**
	Null	9	12		
12	**13**	**14**	**15**	**16**	**17**
11			0A	2	02

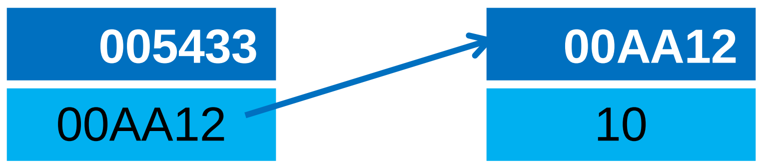


What tree is this?

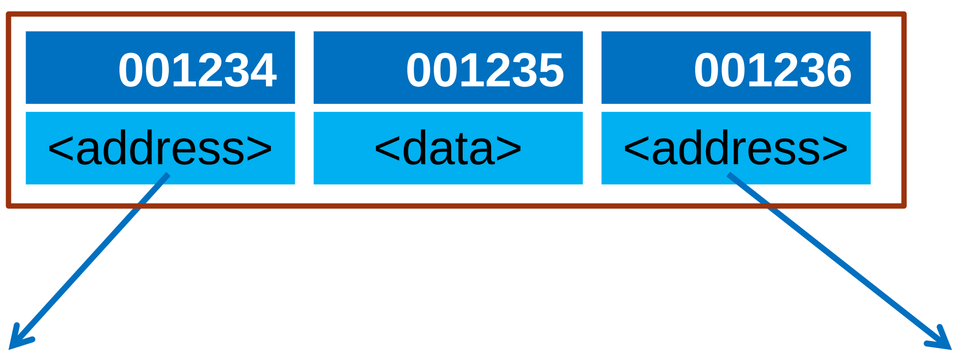
00	**01**	**02**	**03**	**04**	**05**
0A	0D	3	NULL	16	7
06	**07**	**08**	**09**	**0A**	**0B**
NULL	5	13	02	4	10
0C	**0D**	**0E**	**0F**	**10**	**11**
NULL	1	16	07	11	NULL
12	**13**	**14**	**15**	**16**	**17**
NULL	9	NULL	NULL	2	NULL

Ideas

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

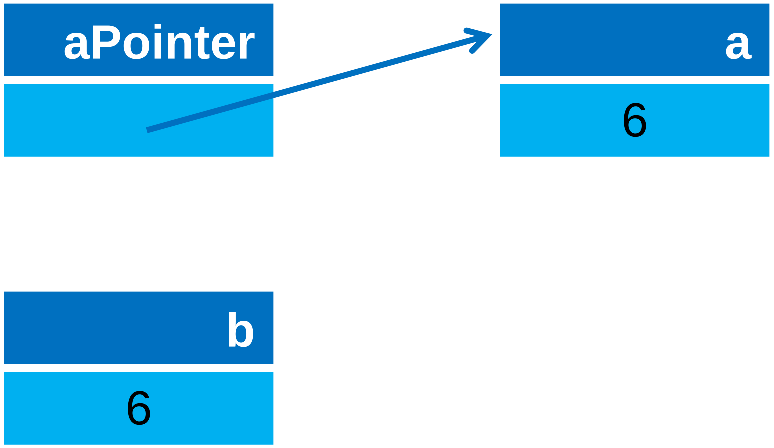


- Using adjacency to create relationship



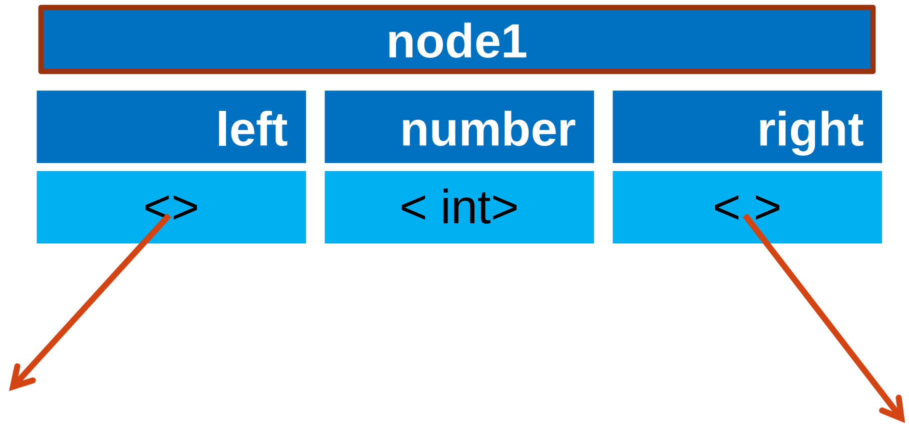
"C" syntax

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

a = 6;
aPointer = &a;

b = *aPointer;
```



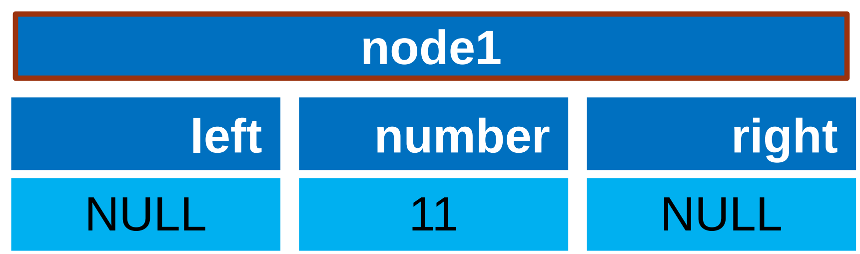
“C” syntax

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

Nodes node1, node2, node3;
```



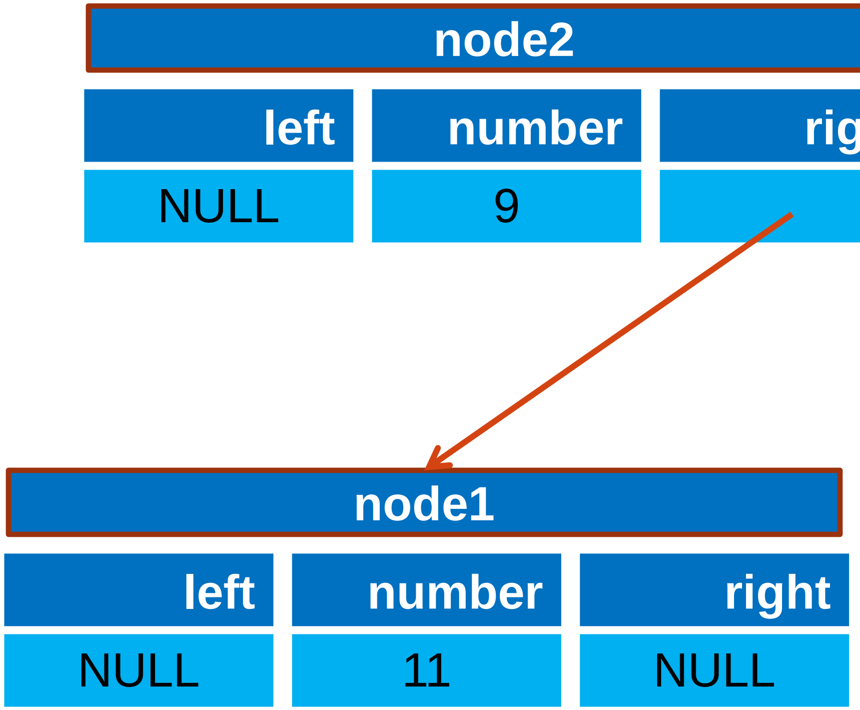
“C” syntax

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

Node node1, node2, node3;

node1.number = 11;
node1.left = NULL;
node1.right = NULL;
```



"C" syntax

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

Node node1, node2, node3;

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



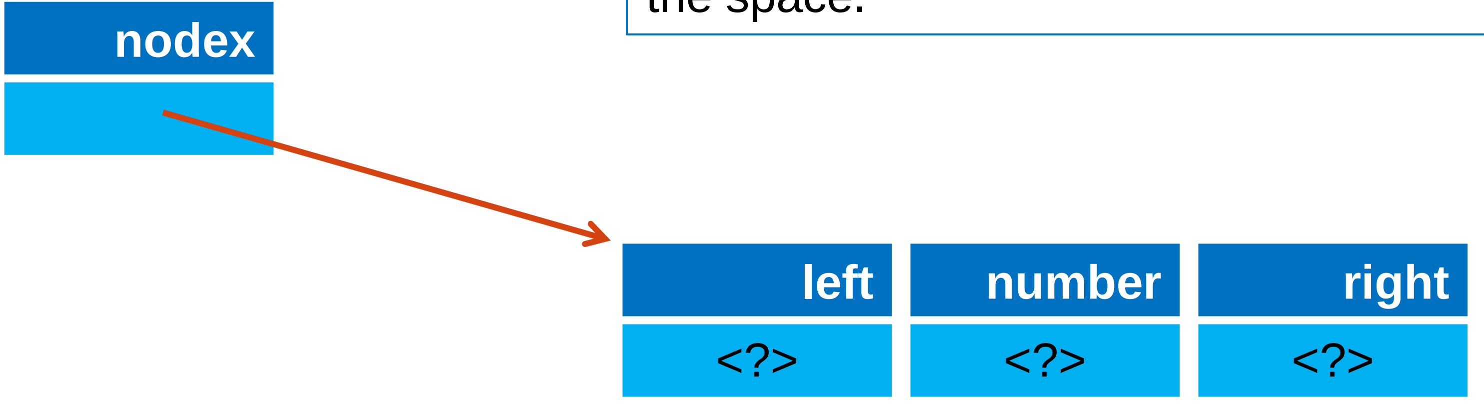
Dynamic memory allocation

Q: Do all the nodes have to be allocated in advance?

A: No. They can be dynamically allocated.

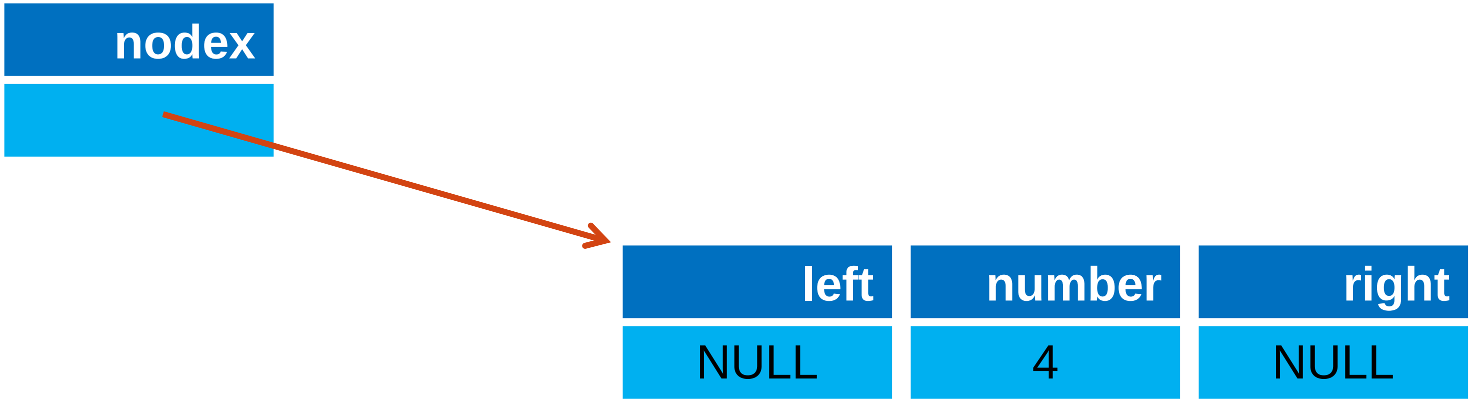
```
Node* nodex;

nodex = malloc (sizeof (Node));
```

Note: there is no "name" for the allocated space; only a pointer to the beginning of the space.

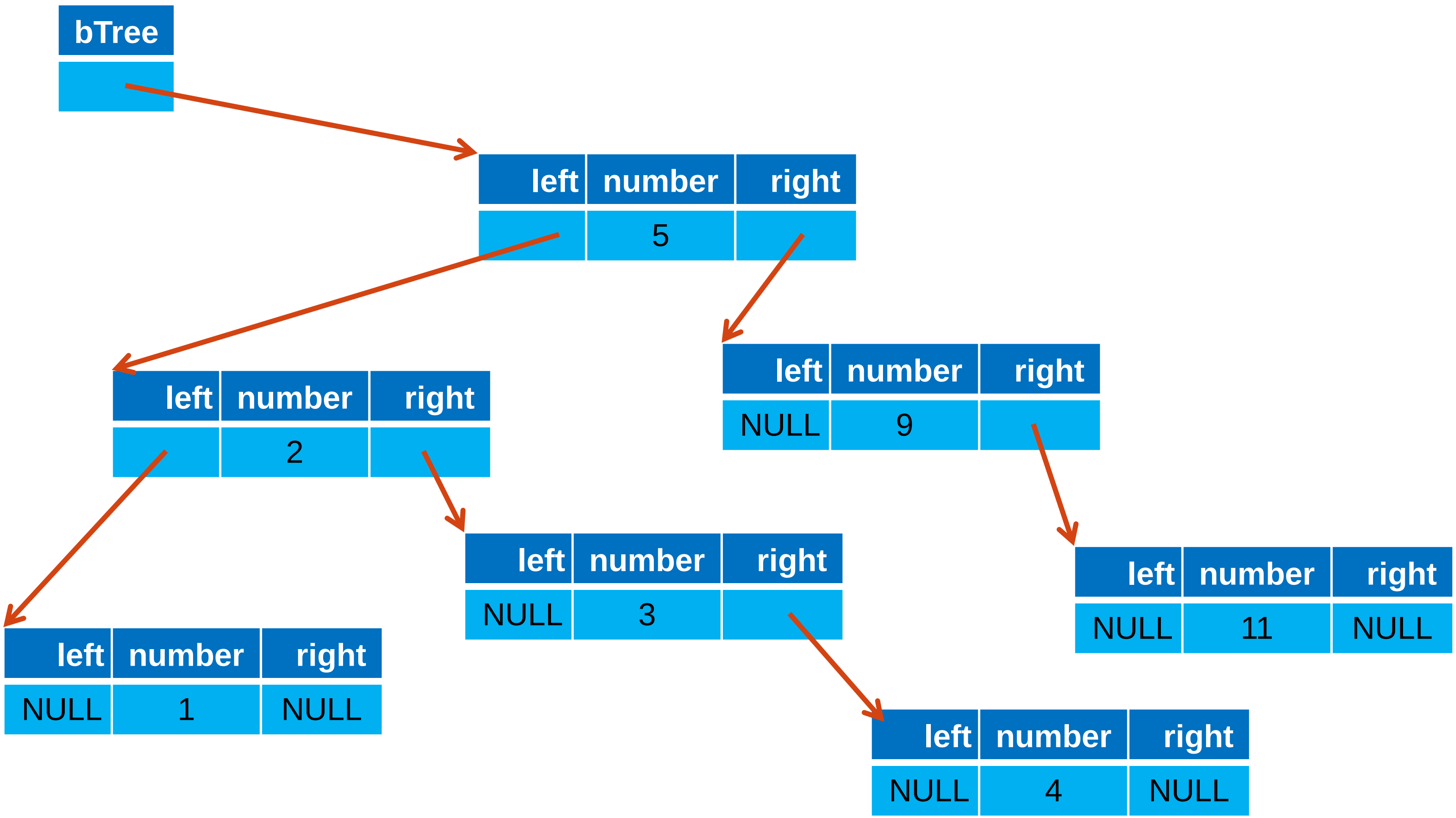


Using dynamic memory

```
Node* nodex;

nodex = malloc( sizeof(Node));
(*nodex).number = 4;
(*nodex).left = NULL;
(*nodex).right = NULL;
```



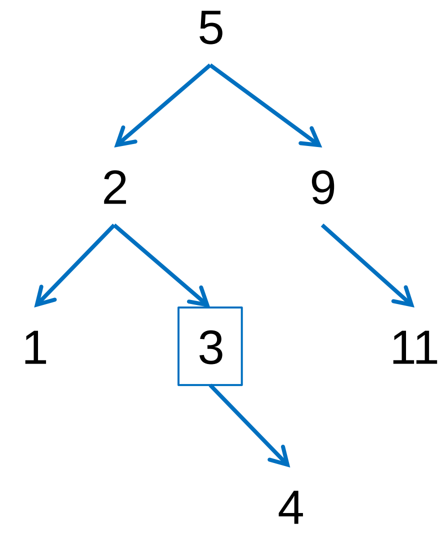
Dynamically created tree



Traversal

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”?

```
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

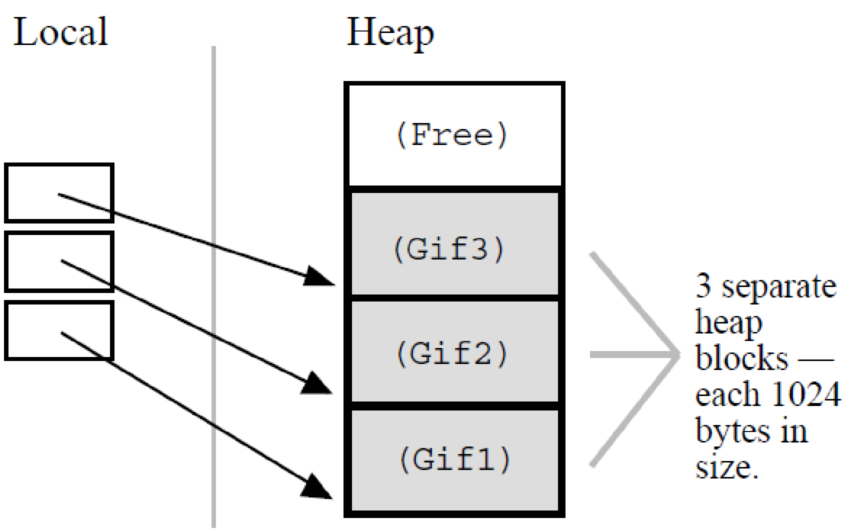
```
void X() {
   int a = 1;
   int b = 2;
   // T1

   Y(a);
   // T3
   Y(b);

   // T5

}

void Y(int p) {
   int q;
   q = p + 2;
   // T2 (first time through), T4 (second time through)
}
```

T1 - X()'s locals have been allocated and given values..	**T2** - Y() is called with p=1, and its locals are allocated. X()'s locals continue to be allocated.	**T3** - Y() exits and its locals are deallocated. We are left only with X()'s locals.	**T4** - Y() is called again with p=2, and its locals are allocated a second time.	**T5** - Y() exits and its locals are deallocated. X()'s locals will be deallocated when it exits.
	Y() p = 1, q = 3		Y() p = 2, q = 4	
X() a = 1, b = 2	X() a = 1, b = 2	X() a = 1, b = 2	X() a = 1, b = 2	X() a = 1, b = 2

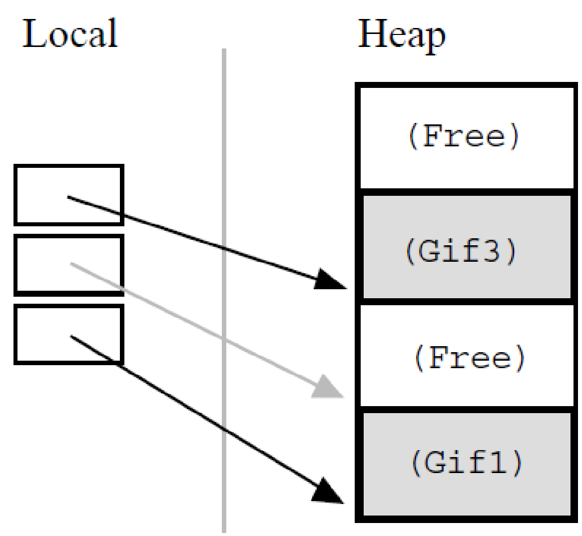
Violation of lifetime

```
// 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



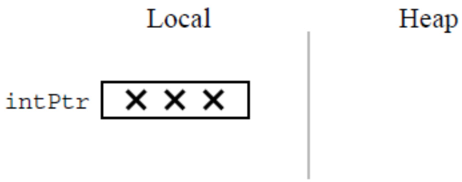
allocation: void* malloc (unsigned long size);



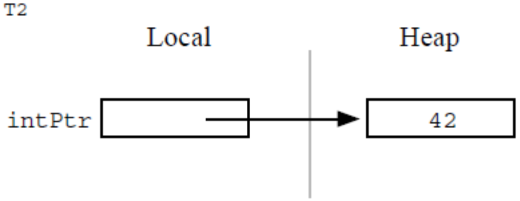
deallocation: void free (void* block);

Heap storage

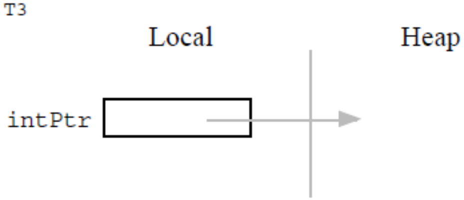
```
void Heap1() {
   int* intPtr;
   // Allocates local pointer local variable (but not its pointee)
   // T1
```




```
   // Allocates heap block and stores its pointer in local variable.
   // Dereferences the pointer to set the pointee to 42.
   intPtr = malloc(sizeof(int));
   *intPtr = 42;
   // T2
```




```
   // Deallocates heap block making the pointer bad.
   // The programmer must remember not to use the pointer
   // after the pointee has been deallocated (this is
   // why the pointer is shown in gray).
   free(intPtr);
   // T3
```




```
}
```

Heap storage

```
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 mangement 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)

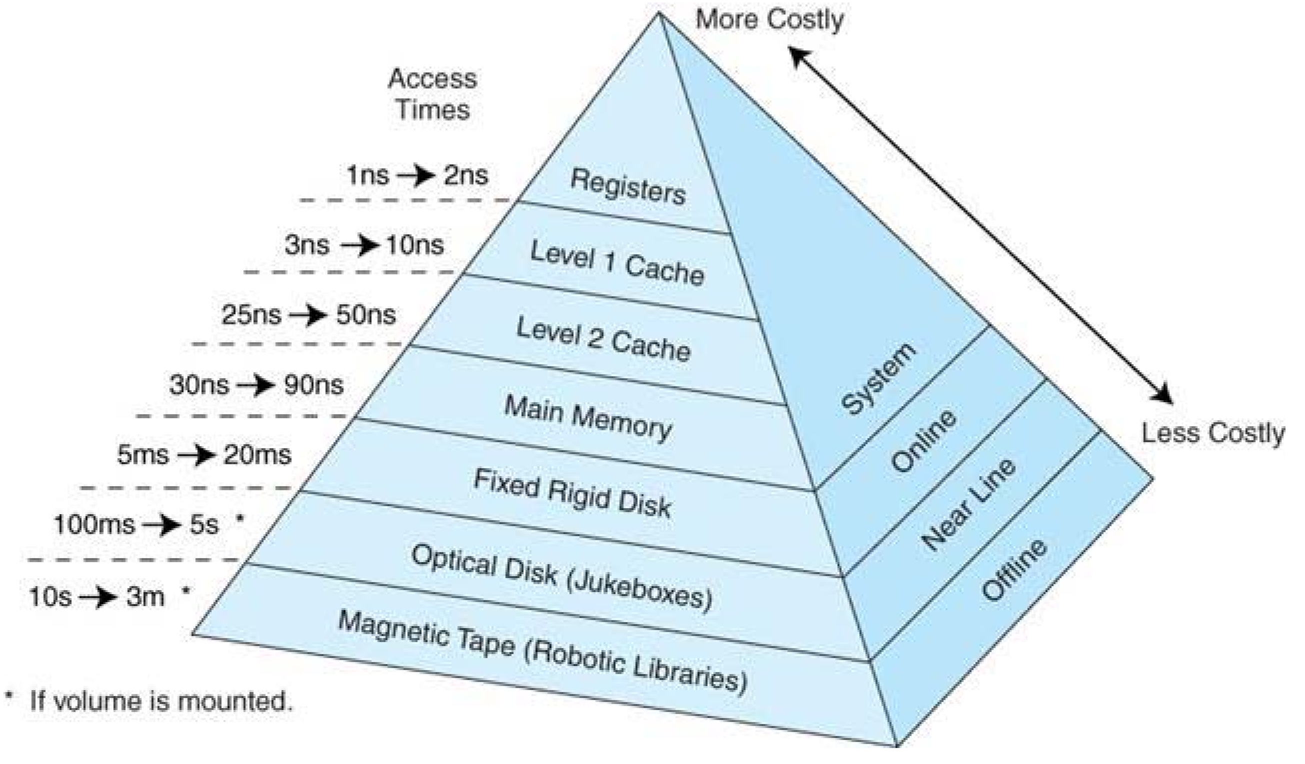
Mark and sweep

- 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 hierarcy



Memory hierarchy

- Why does this matter? To algorithm design?

Memory	Access Time	Normalized Human	Location
Register	1 nsec	1 sec	on desk
Cache	20 nsec	20 sec	in room
Main Memory	50 nsec	1 minute	next door
Disk	10 msec	100 days	off planet