Exercise Session 4

Systems Programming and Computer Architecture

Fall Semester 2024

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**: they are there to complement and go into more depth where I found appropriate
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Agenda

- Review of **assignment 2**
  - Pointers
  - BSTs
- **Quiz** – Test your understanding of C code
- Preview of **assignment 3**: malloclab
  - The task
  - **Recap:** Dynamic memory allocation
  - Some tips

Last Week’s Assignment

C Programming

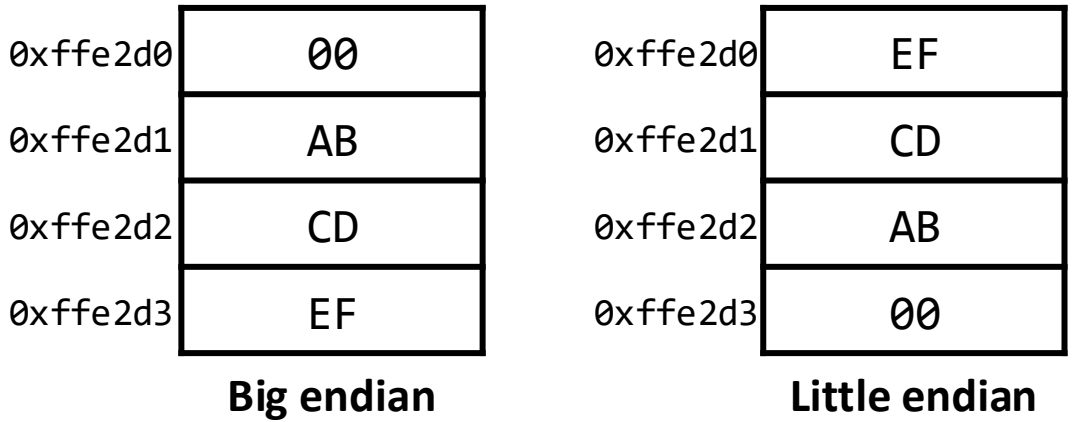
Assignment 2

Always check for **`ptr != NULL`** before dereferencing (attempting to use `*ptr` to access the address pointed to by the pointer)

Task 3 – Little vs. big endian

Write a C program that prints out whether the computer it is running on is little endian or big endian. (hint: pointer and casts)

```
a = 0xABCDEF
```



Task 3 – Buggy Solution 1

```c
#include <stdio.h>

int get_endian() {
  int x = 1;
  char *charptr = (char *) &x;

  return (*charptr != "0x01");
}

int main(void) {
  get_endian();
  printf("little endian? %s\n", get_endian ? "true" : "false");
  return 0;
}
```

Task 3 – Buggy Solution 1

```c
#include <stdio.h>

int get_endian() {
  int x = 1;
  char *charptr = (char *) &x;

  return (*charptr != "0x01");
}

int main(void) {
  get_endian();
  printf("little endian? %s\n", get_endian ? "true" : "false");
  return 0;
}
```

- Here we compare the value of the **byte** pointed to by charptr (value `0x1`) with the address of the string literal "0x01“
- In C, memory addresses are stored as integer numbers of type `unsigned long int` or `uintptr_t` – that is why this comparison is legal
- This logical expression would only evaluate as true if the address of the string literal was 1

Task 3 – Buggy Solution 1

```c
#include <stdio.h>

int get_endian() {
  int x = 1;
  char *charptr = (char *) &x;

  return (*charptr != "0x01");
}

int main(void) {
  get_endian();
  printf("little endian? %s\n", get_endian ? "true" : "false");
  return 0;
}
```

- Here we use the address of the function `get_endian` as the logical expression of a conditional expression.
- Because a function's address is never 0, this logical expression always evaluates as true.

Task 3 – Buggy Solution 1

```c
#include <stdio.h>

int get_endian() {
  int x = 1;
  char *charptr = (char *) &x;

  return (*charptr != "0x01");
}

int main(void) {
  get_endian();
  printf("little endian? %s\n", get_endian ? "true" : "false");
  return 0;
}
```

Reminder: The compiler is your friend!

```
gcc test.c -Wall
test.c: In function ‘get_endian’:
test.c:7:22: warning: comparison between pointer and
integer
    7 |         return (*charptr != "0x01");
      |                          ^~
test.c:7:22: warning: comparison with string literal
results in unspecified behavior [-Waddress]
test.c: In function ‘main’:
test.c:12:46: warning: the address of ‘get_endian’ will
always evaluate as ‘true’ [-Waddress]
   13 |           get_endian ? "true" : "false");
      |                        ^
```

Task 3 – Buggy Solution 2

```c
#include <stdio.h>

void print_endian() {
  int x = 1;
  char *charptr = (char *) &x;

  if (charptr != 0)
    printf("little endian\n");
  else
    printf("big endian\n");
}

int main(void) {
  print_endian();
  return 0;
}
```

Task 3 – Buggy Solution 2

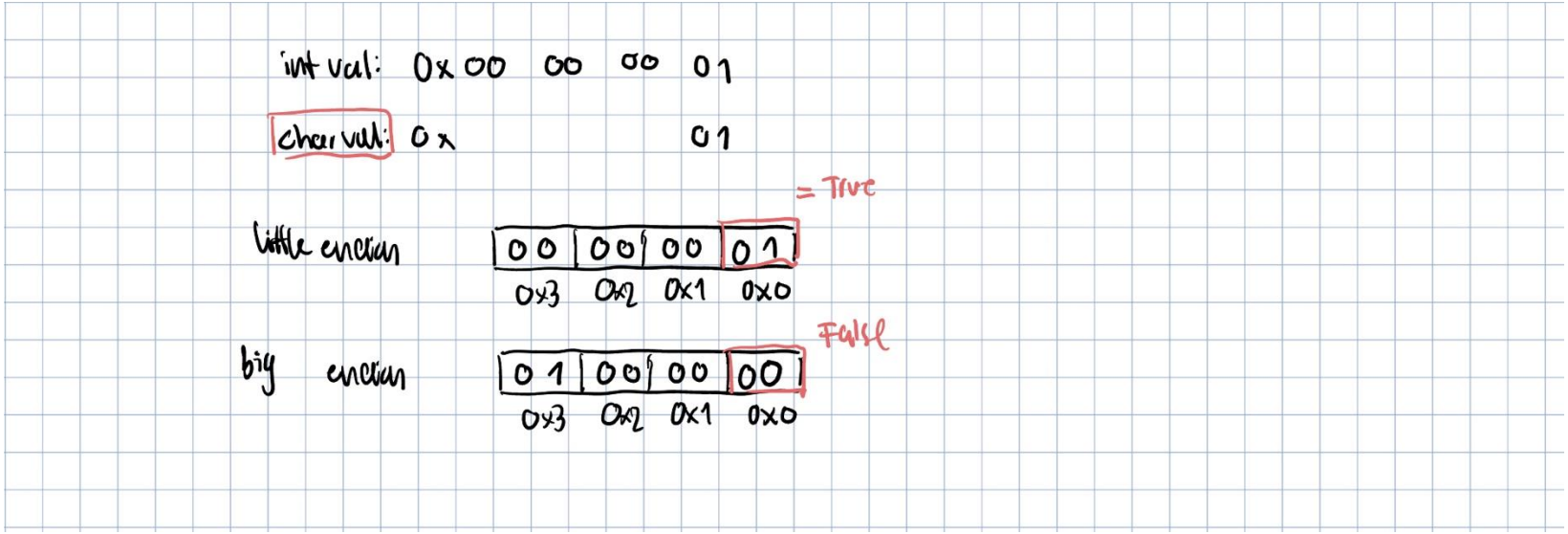
```c
#include <stdio.h>

void print_endian() {
  int x = 1;
  char *charptr = (char *) &x;

  if (charptr != 0)
    printf("little endian\n");
  else
    printf("big endian\n");
}

int main(void) {
  print_endian();
  return 0;
}
```

- Here we compare the value of **the pointer** with 0.
- This comparison will never be true, as the address of a local variable cannot be 0.

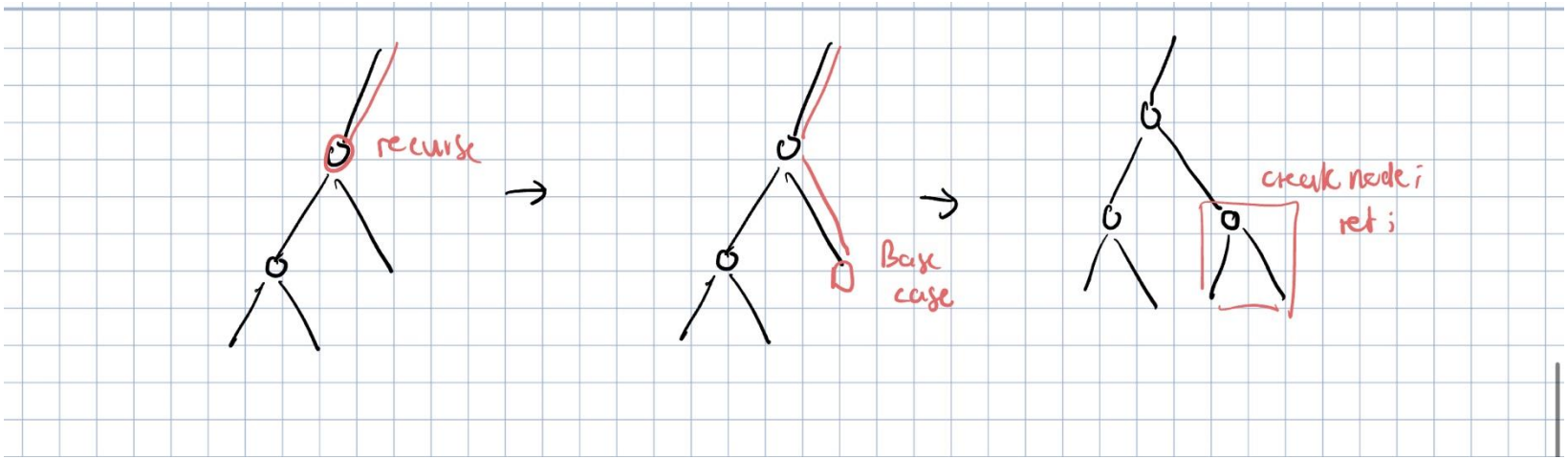
Task 3 – Solution

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int x = 1;
  char *charptr = (char *) &x;

  if (*charptr == 1)
    printf("little endian\n");
  else
    printf("big endian\n");

  return 0;
}
```

Task 3: How does that look like?



Task 6: Quick walkthrough: Insert



Task 6: Quick walkthrough: Insert

Systems@ETH Zürich

```
3  tree_node* insert(tree_node* root, int key, int value) {
4    if(root==NULL){
5      tree_node* newroot = (tree_node*) malloc(sizeof(tree_node));
6      newroot->key=key;
7      newroot->value=value;
8      newroot->left=NULL;
9      newroot->right=NULL;
10     return newroot;
11   }else{
12     if(key<root->key) root->left = insert(root->left, key, value);
13     if(key>root->key) root->right = insert(root->right, key, value);
14     if(key==root->key) root->value=value;
15     return root;
16   }
17 }
```

Task 6: Quick walkthrough: Lookup

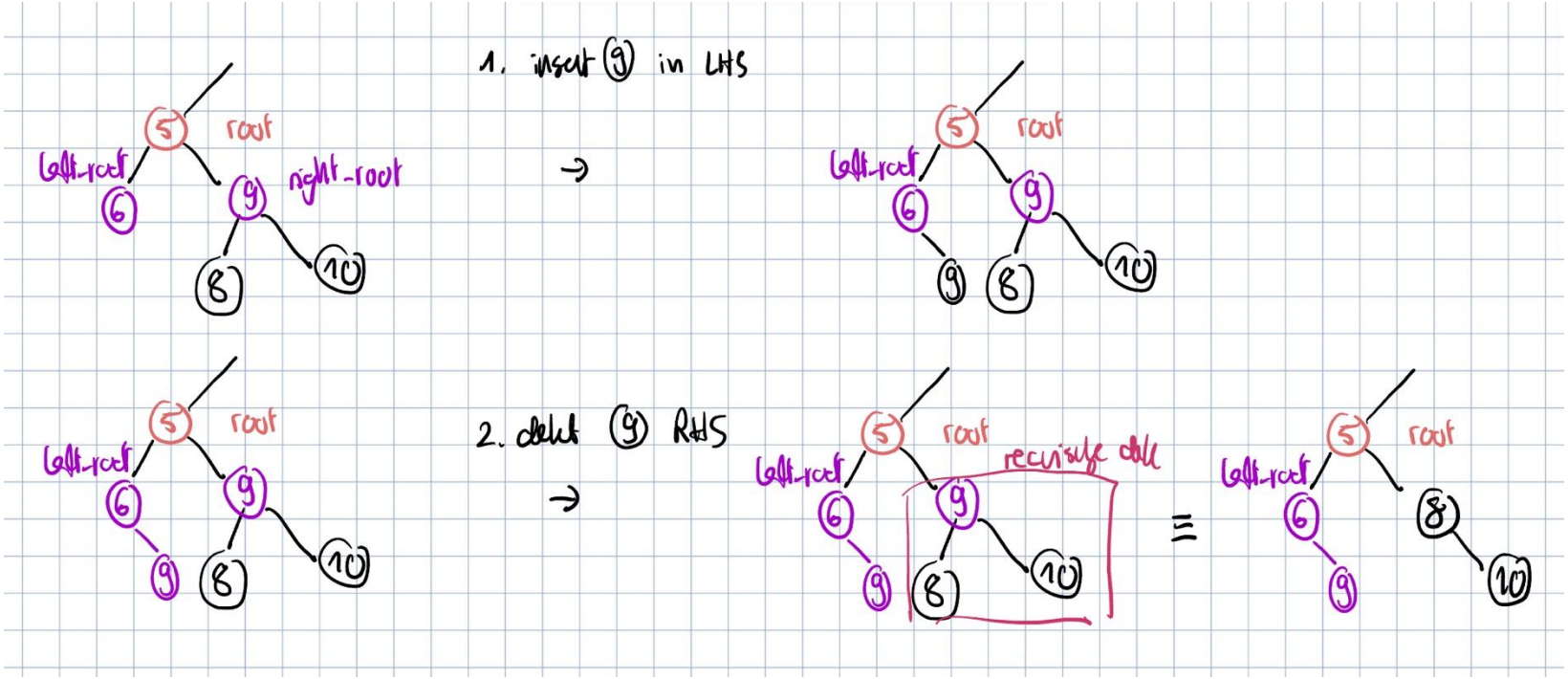
Systems@ETH Zürich

```
tree_node* lookup(tree_node* root, int key) {
  if(root==NULL) return NULL;
  if(key<root->key) return lookup(root->left, key);
  if(key>root->key) return lookup(root->right, key);
  if(key==root->key) return root;
}
```

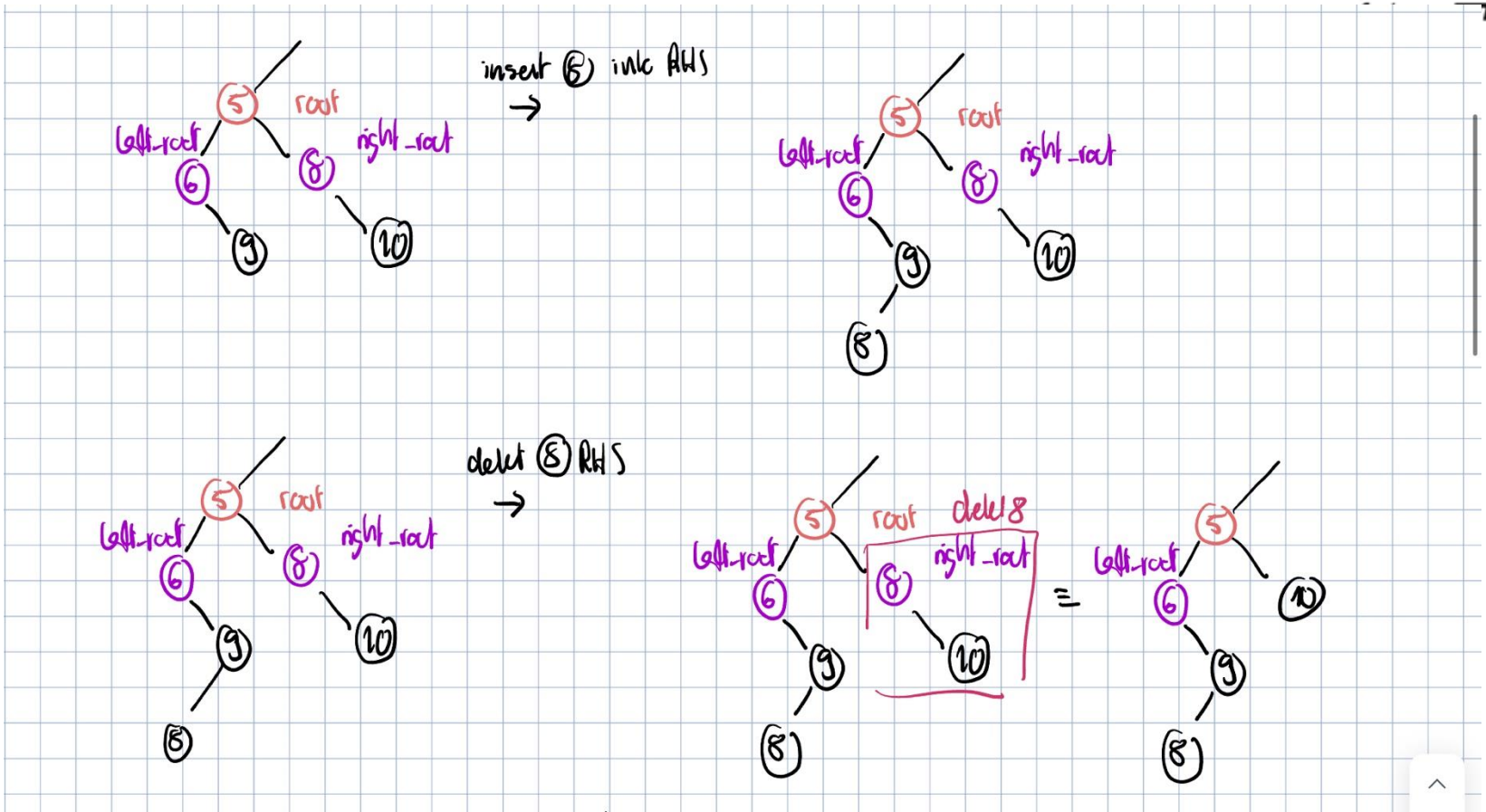
Task 6: Remark Master solution (delete)

- I know this is no algorithms course, yet it's a **greatly** inefficient solution
- What is it doing? Shoveling all nodes from the right subtree into the left subtree

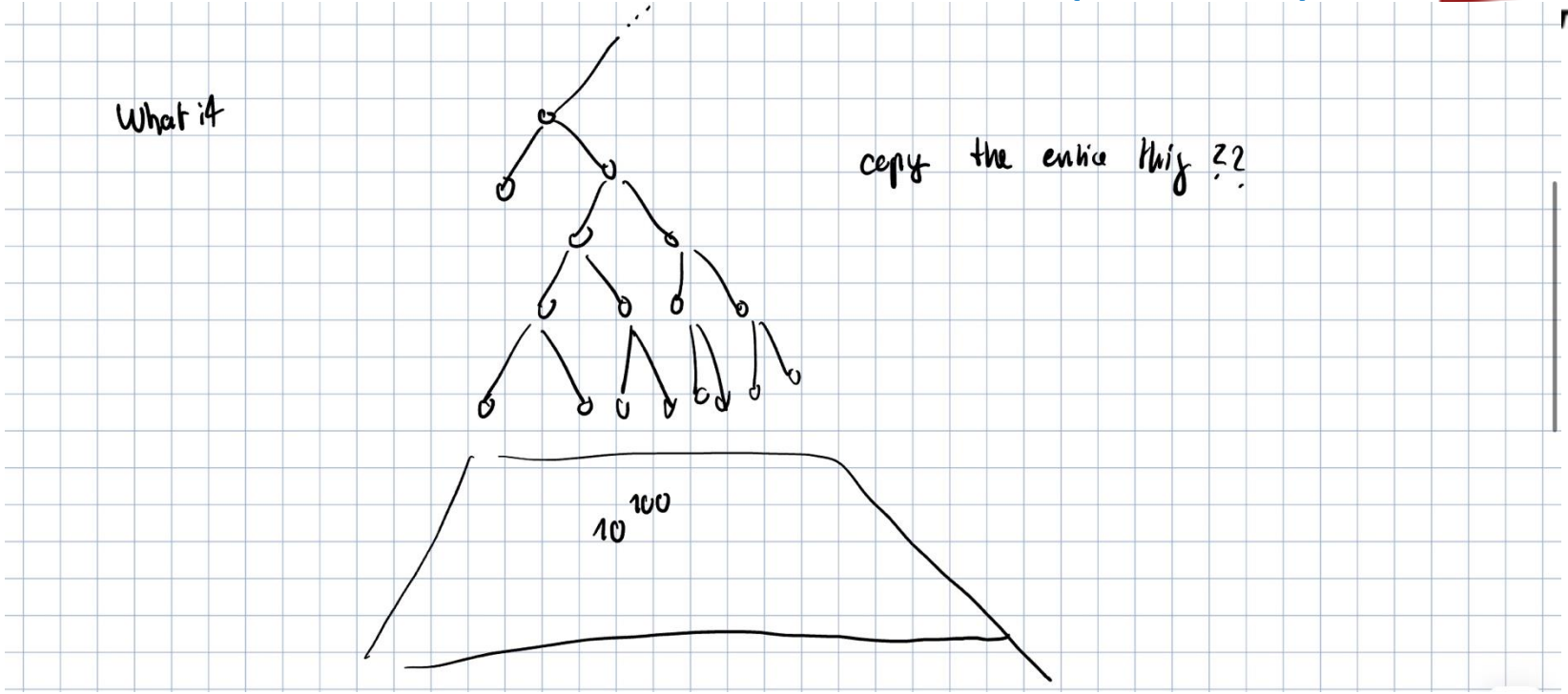
Task 6: Remark Master solution (delete)



Task 6: Remark Master solution (delete)

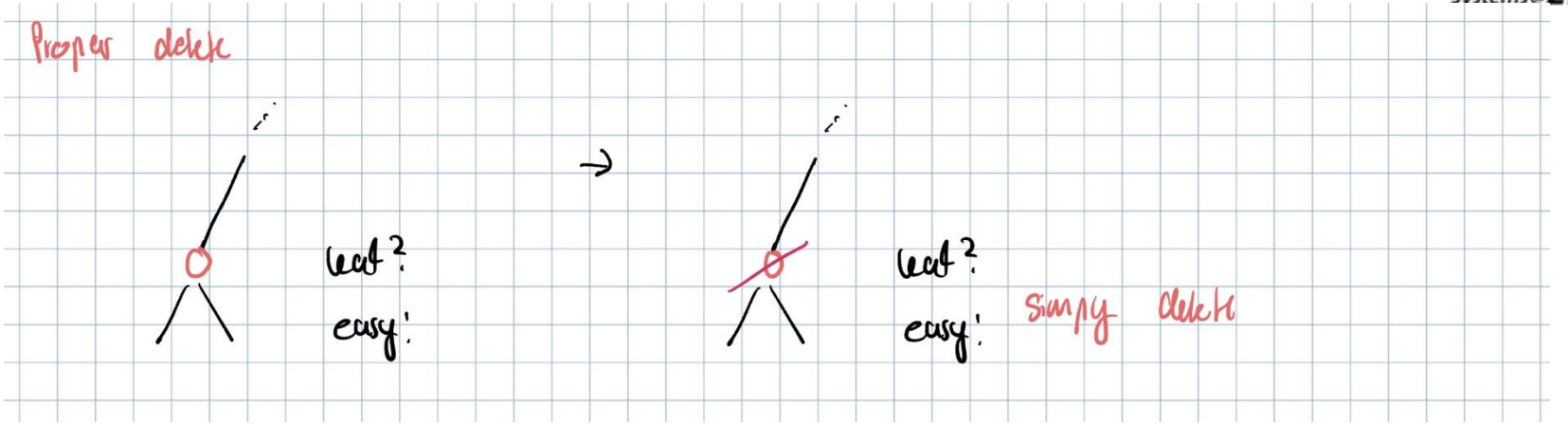


Task 6: Remark Master solution (delete)



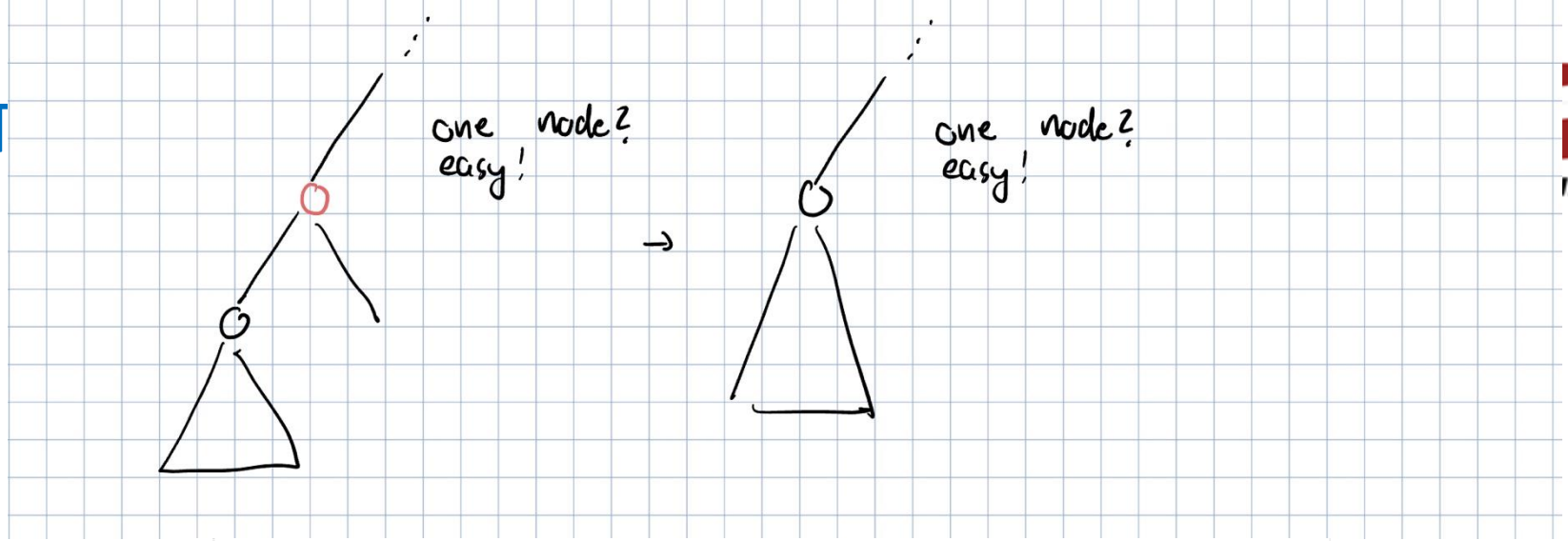
- Depending on test cases in the exam **THIS WILL TIME OUT**

Task 6: Quick walkthrough: Delete (proper)



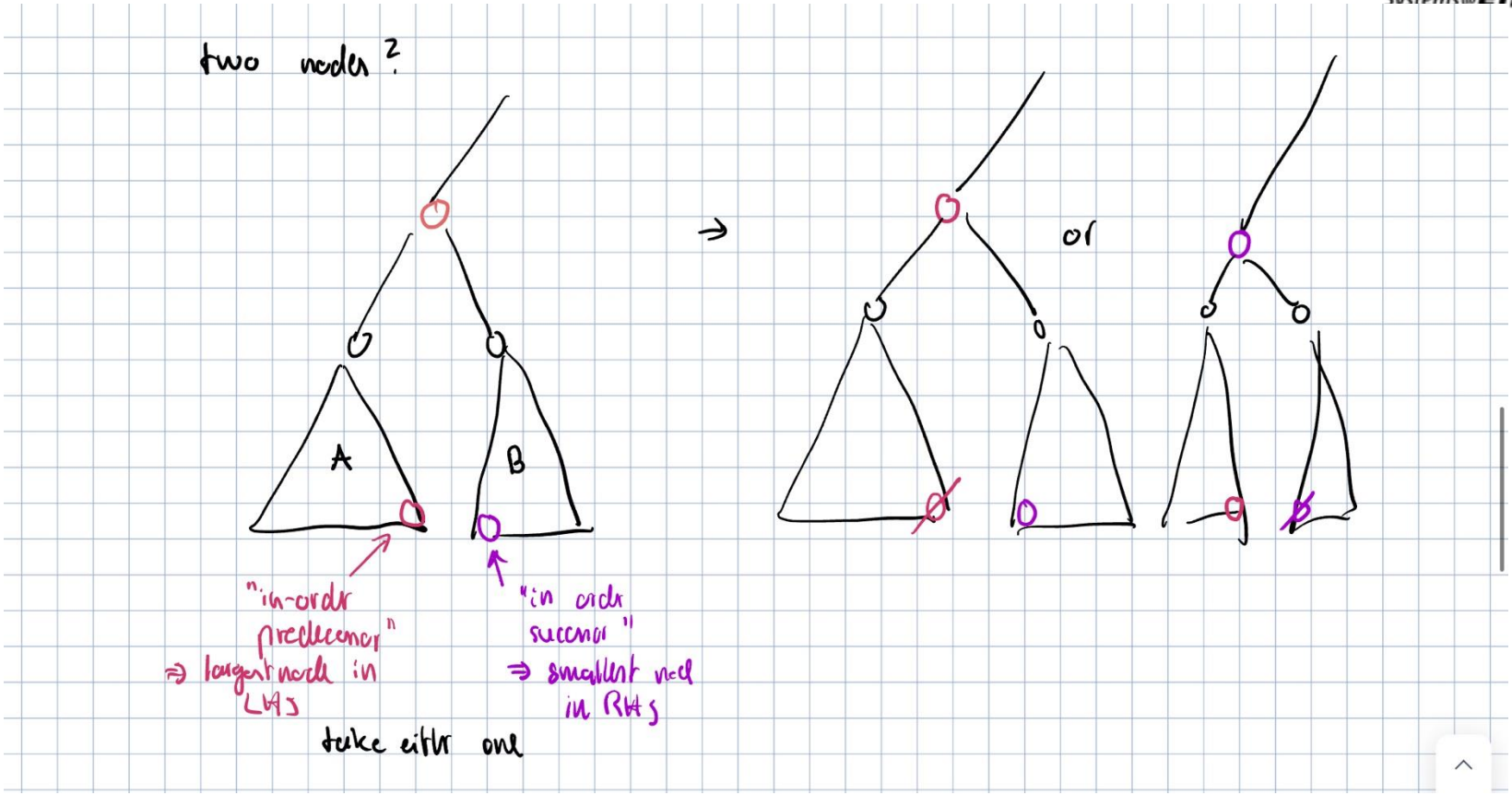

```c
//leaf
if(root->left==NULL && root->right==NULL){
  free(root);
  return NULL;
}
```

T




```c
//one child
if(root->left==NULL && root->right!=NULL){
  tree_node* temp = root->right;
  free(root);
  return temp;
}

if(root->left!=NULL && root->right==NULL){
  tree_node* temp = root->left;
  free(root);
  return temp;
}
```

Task 6: Quick walkthrough: Delete (proper)

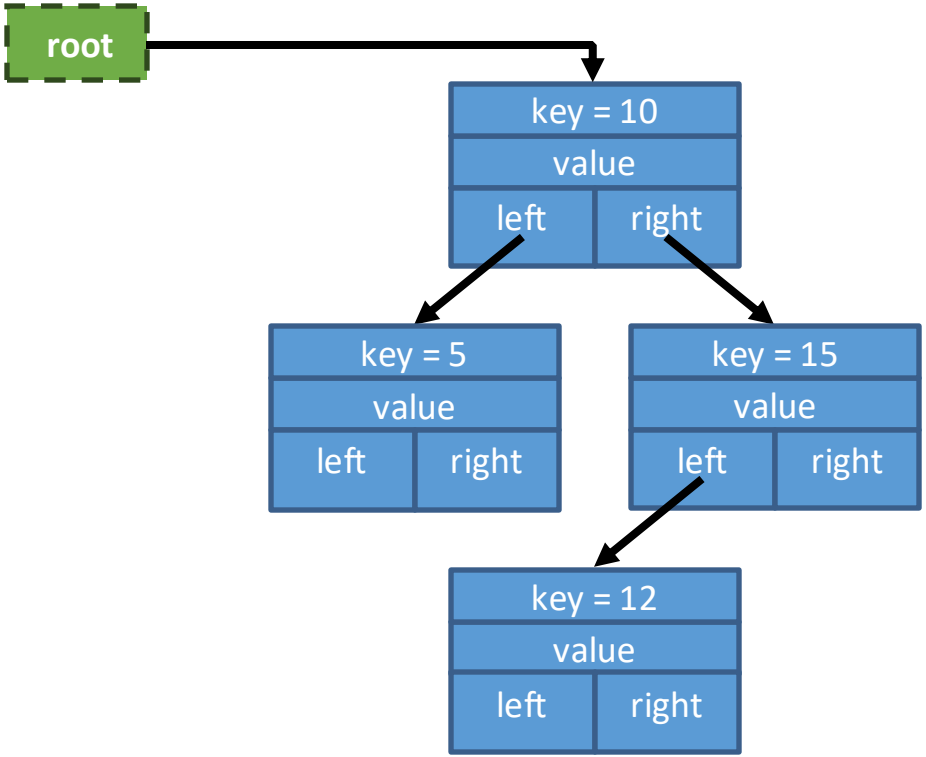


Task 6: Quick walkthrough: Delete (proper)

Systems@ETH Zürich

```
//two children
tree_node* succ = root->left;
tree_node* succ_parent = root;

while (succ->right != NULL) {
    succ_parent = succ;
    succ = succ->right;
}

root->key = succ->key;
root->value = succ->value;

if (succ_parent == root) {
    succ_parent->left = succ->left;
} else {
    succ_parent->right = succ->left;
}

free(succ);
```

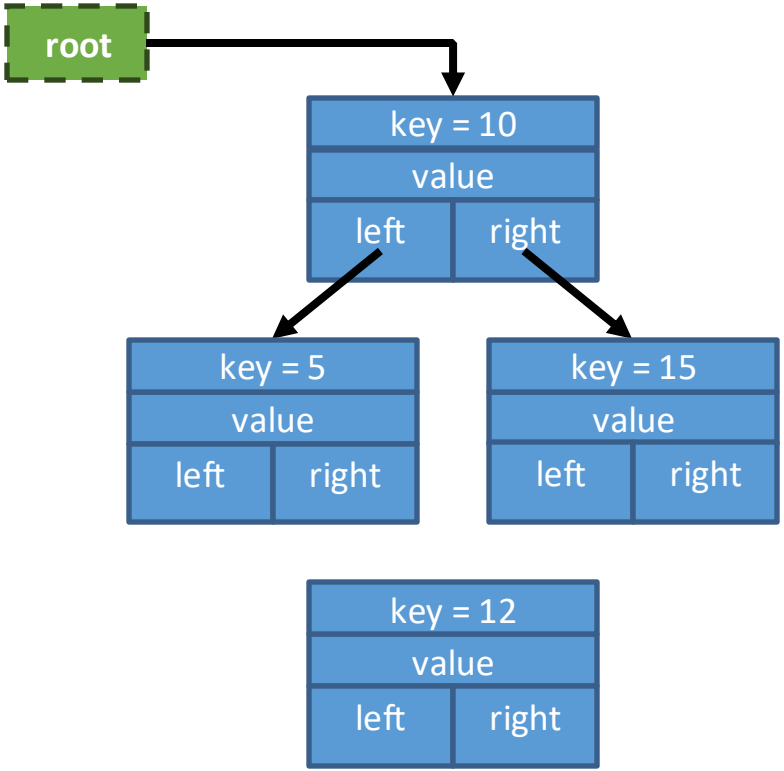
Task 6: Binary Search Tree (BST)



The representation of a tree node in C:

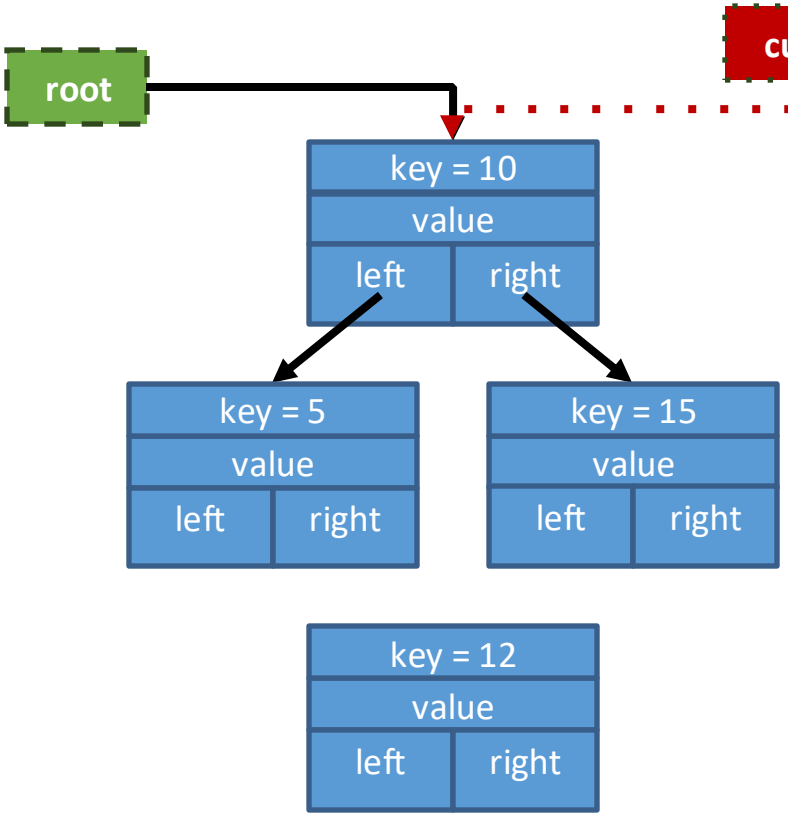
```c
typedef struct tree_node {
  int key;
  int value;
  struct tree_node *left;
  struct tree_node *right;
} tree_node;
```

Task 6: Insert – first idea



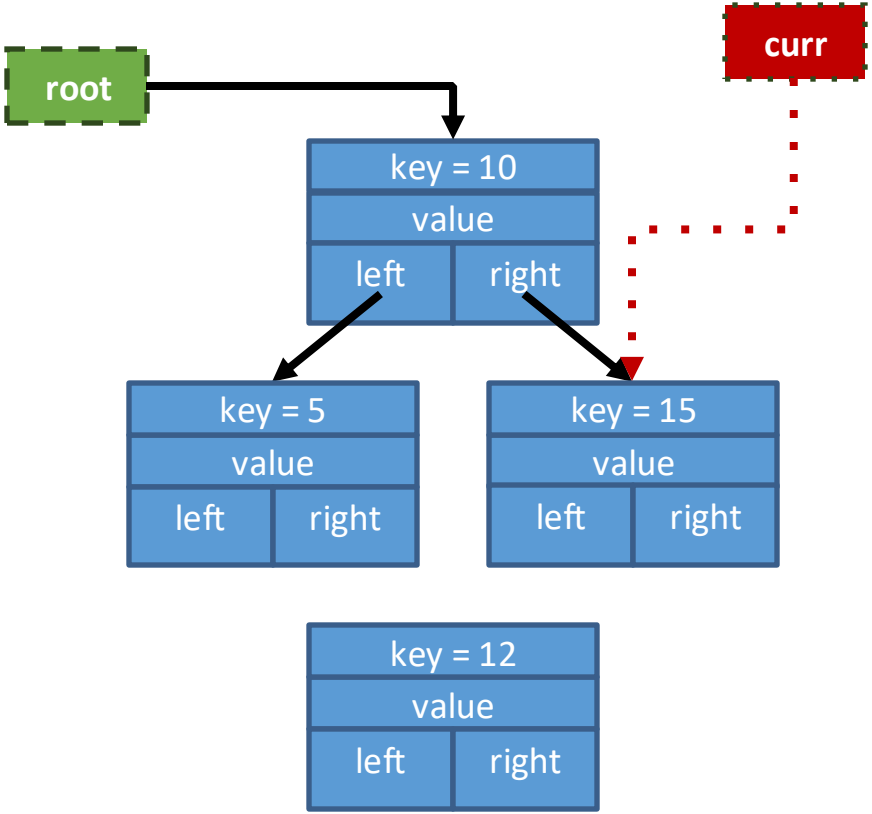

```
tree_node* insert(tree_node *curr, int key, int value)
{
  if(curr == NULL)
                        // place to insert the new node
                        ...

  if(curr->key > key)
                        insert(curr->left, key, value);
  else if(curr->key < key)
                        insert(curr->right, key, value);

  return NULL;
}

insert(root, 12, 42);
```

Task 6: Insert – first idea



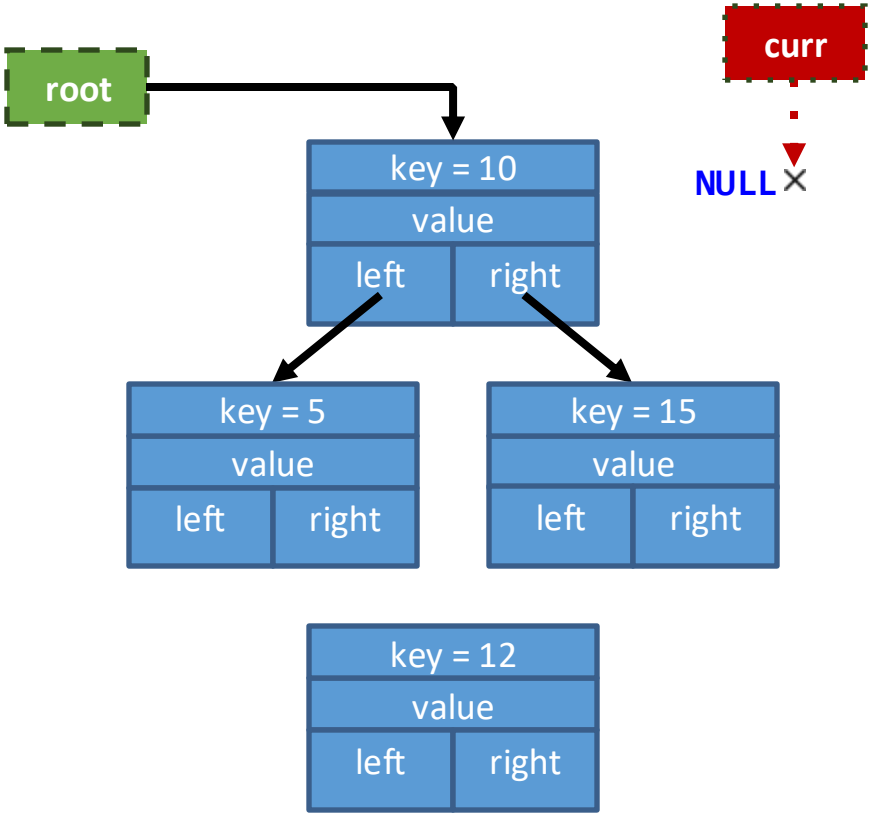

```
tree_node* insert(tree_node *curr, int key, int value)
{
  if(curr == NULL)
                        // place to insert the new node
                        ...

  if(curr->key > key)
                        insert(curr->left, key, value);
  else if(curr->key < key)
                        insert(curr->right, key, value);

  return NULL;
}

insert(root, 12, 42);
```

Task 6: Insert – first idea




```
tree_node* insert(tree_node *curr, int key, int value)
{
  if(curr == NULL)
                          // place to insert the new node
                          ...

  if(curr->key > key)
                          insert(curr->left, key, value);
  else if(curr->key < key)
                          insert(curr->right, key, value);

  return NULL;
}

insert(root, 12, 42);
```

Task 6: Insert – first idea



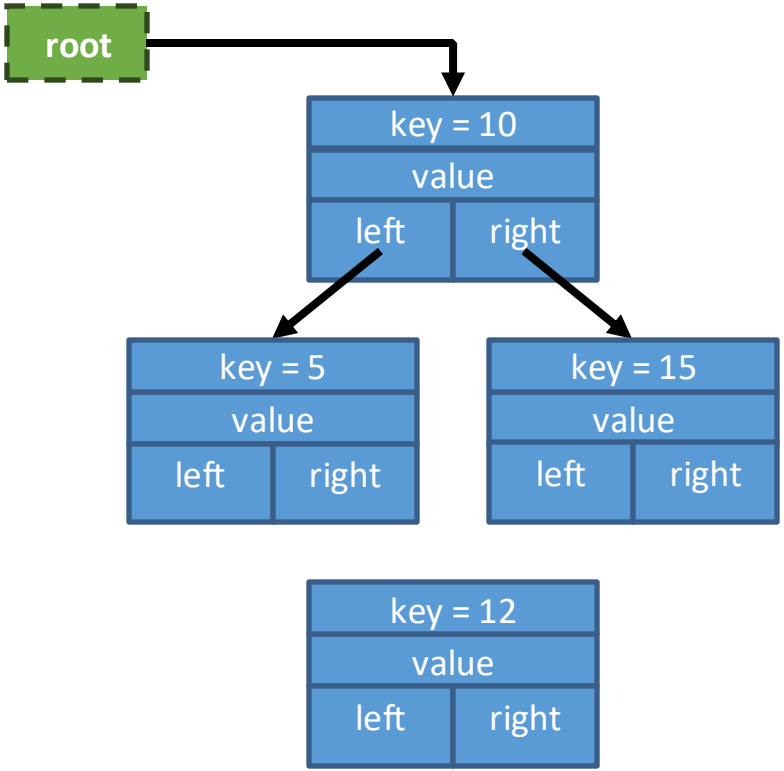
We lost the connection to the tree…

```
tree_node* insert(tree_node *curr, int key, int value)
{
  if(curr == NULL)
                        // place to insert the new node
                        ...

  if(curr->key > key)
                        insert(curr->left, key, value);
  else if(curr->key < key)
                        insert(curr->right, key, value);

  return NULL;
}

insert(root, 12, 42);
```

Issue here?

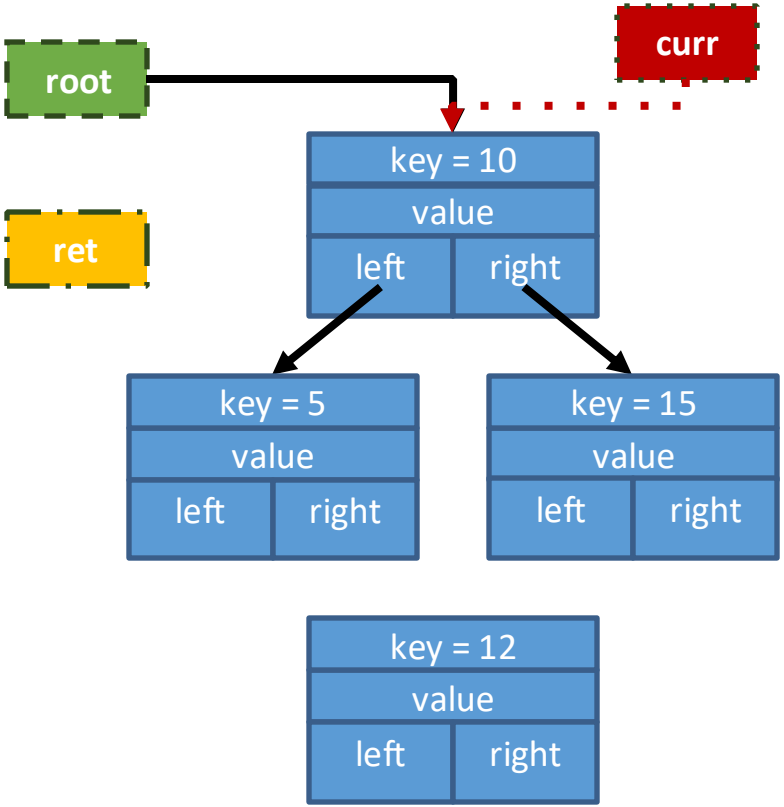
- Now current points to **NULL** and now what?
- Our predecessor didn’t remember us (via cur->left = insert) and we did not remember him (by the function definition and the inherent recursive nature of the program itself)

Task 6: Insert – use the return value



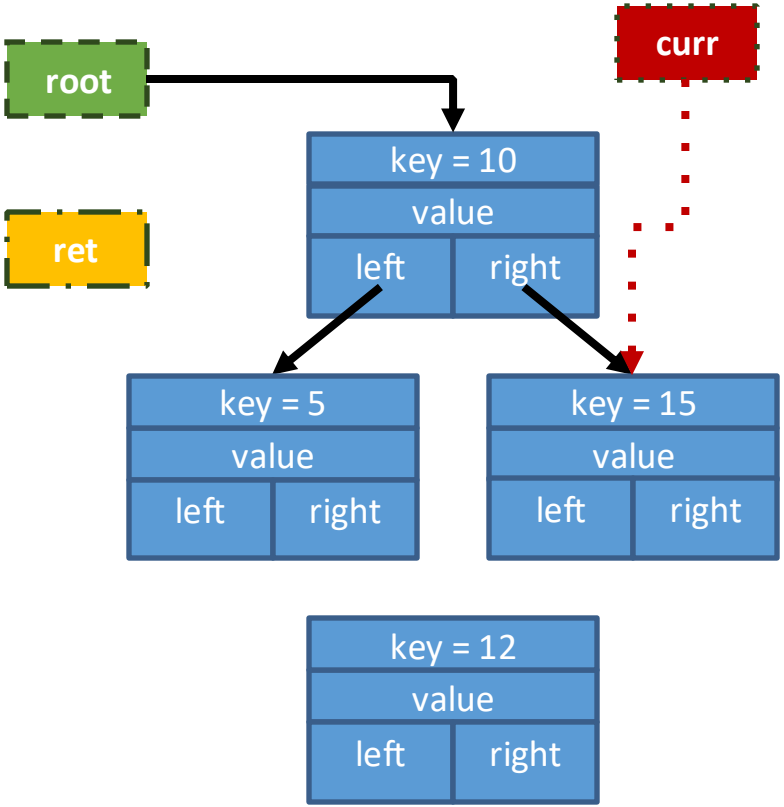

```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6: Insert – use the return value



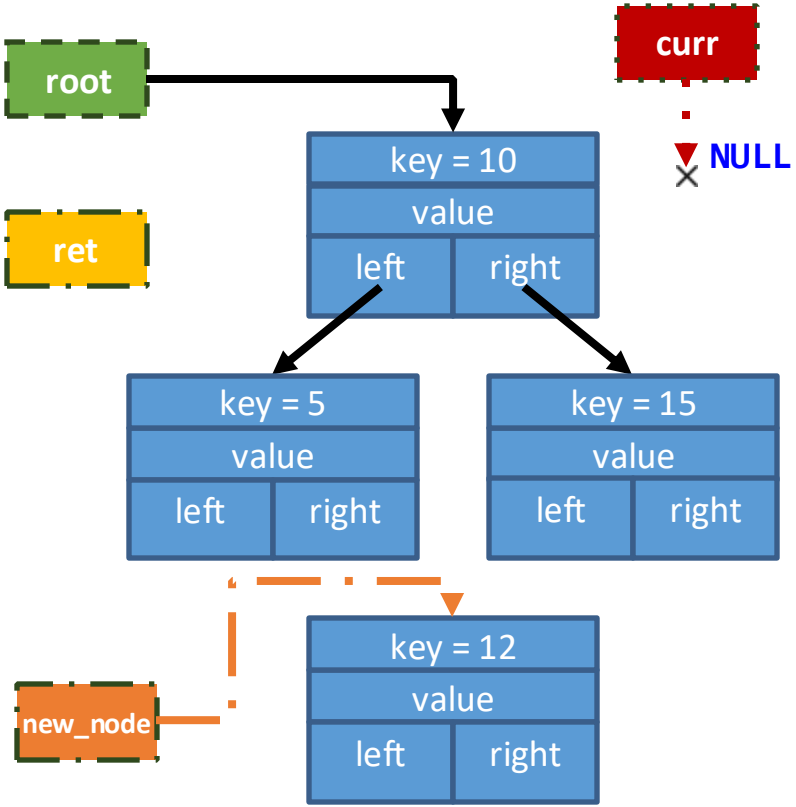

```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6: Insert – use the return value



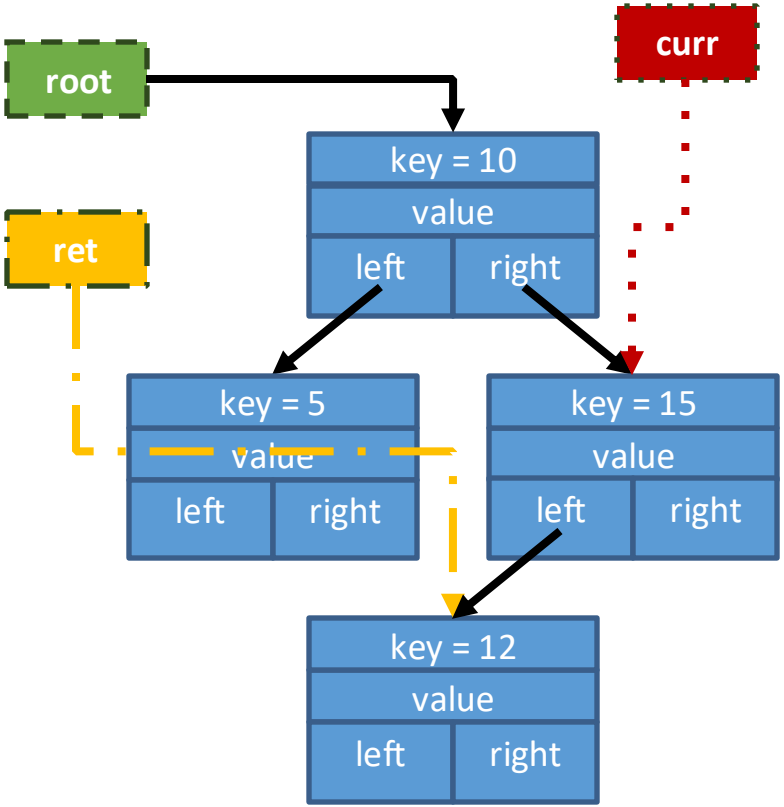

```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6: Insert – use the return value



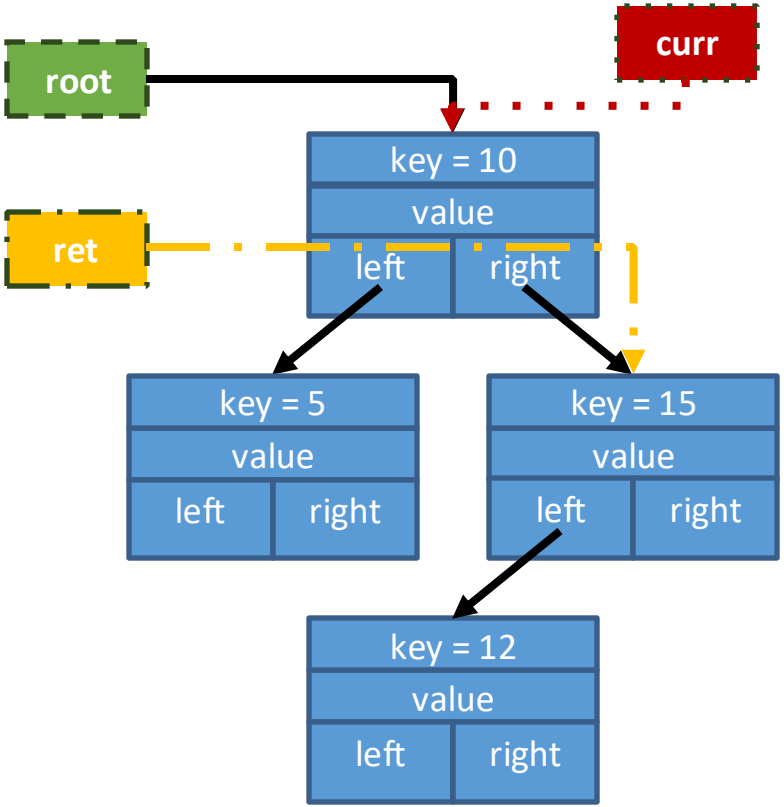

```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6: Insert – use the return value



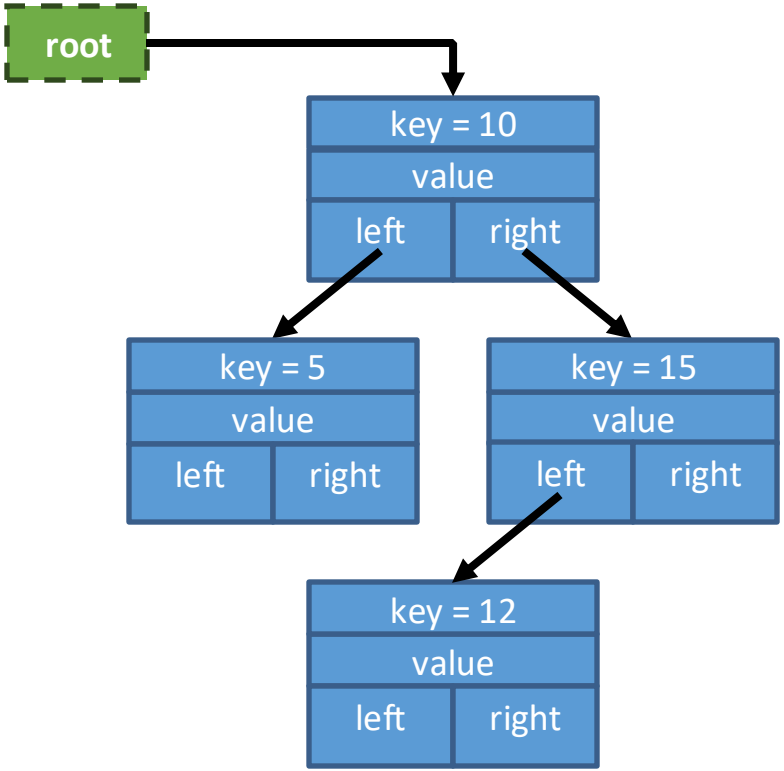

```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6: Insert – use the return value



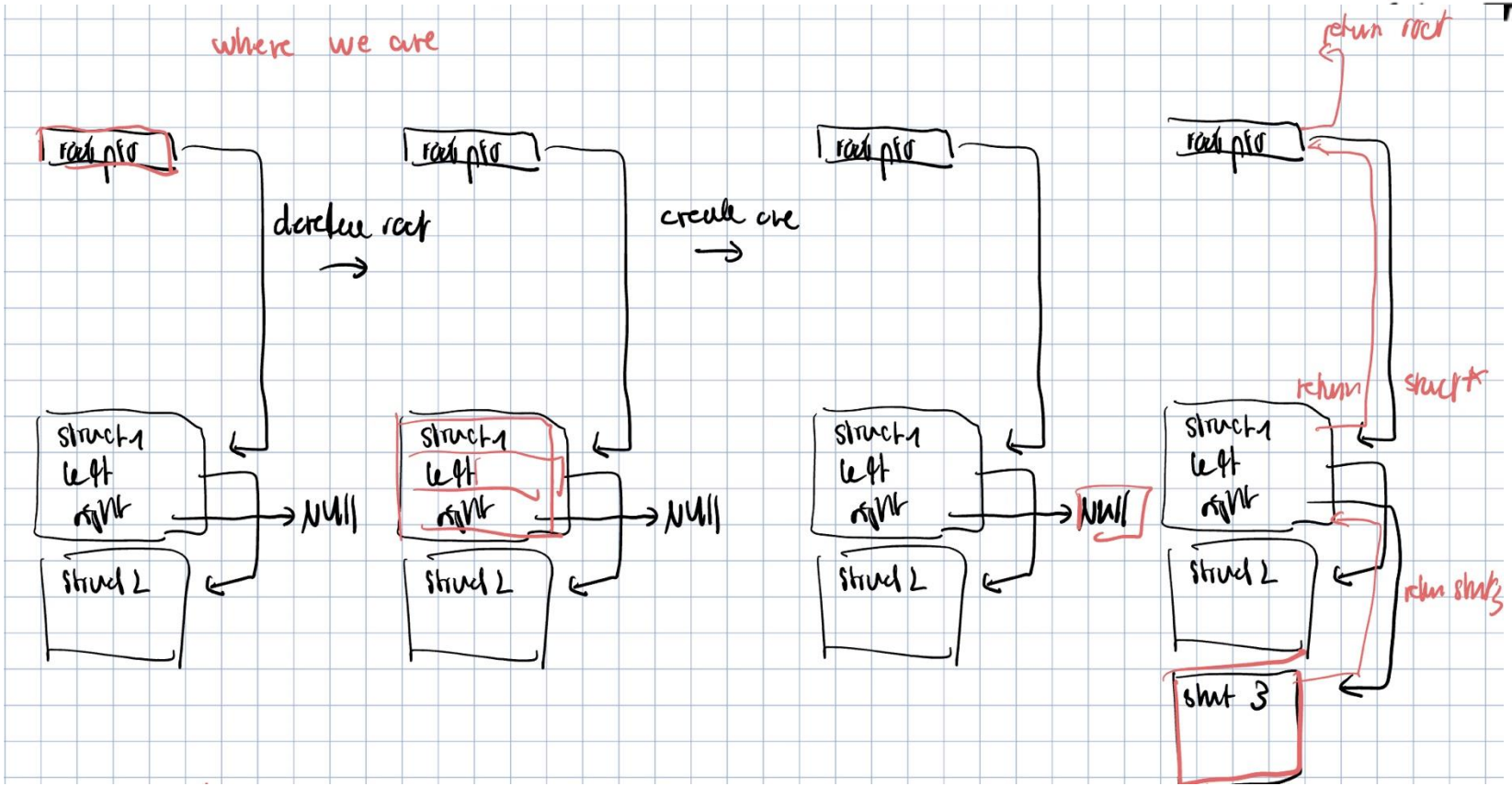

```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6: Insert – use the return value

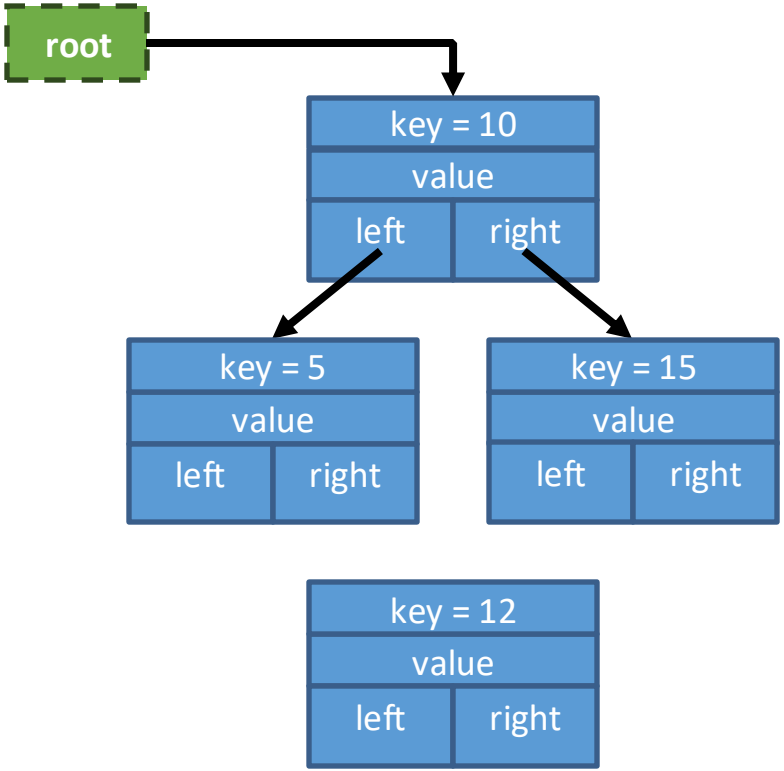



```
tree_node* insert(tree_node *curr, int key, int value) {
  if(curr == NULL) {
            tree_node* new_node = malloc(sizeof(tree_node));
            new_node->key = key;
            new_node->value = value;
            new_node->left = NULL;
            new_node->right = NULL;
            return new_node;
  }

  tree_node* ret;
  if(curr->key > key)
                              curr->left = ret = insert(curr->left, ke
value);
  else if (curr->key < key)
                              curr->right = ret = insert(curr->right,
value);

  return curr;
}

root = insert(root, 12, 42);
```

Task 6 Before



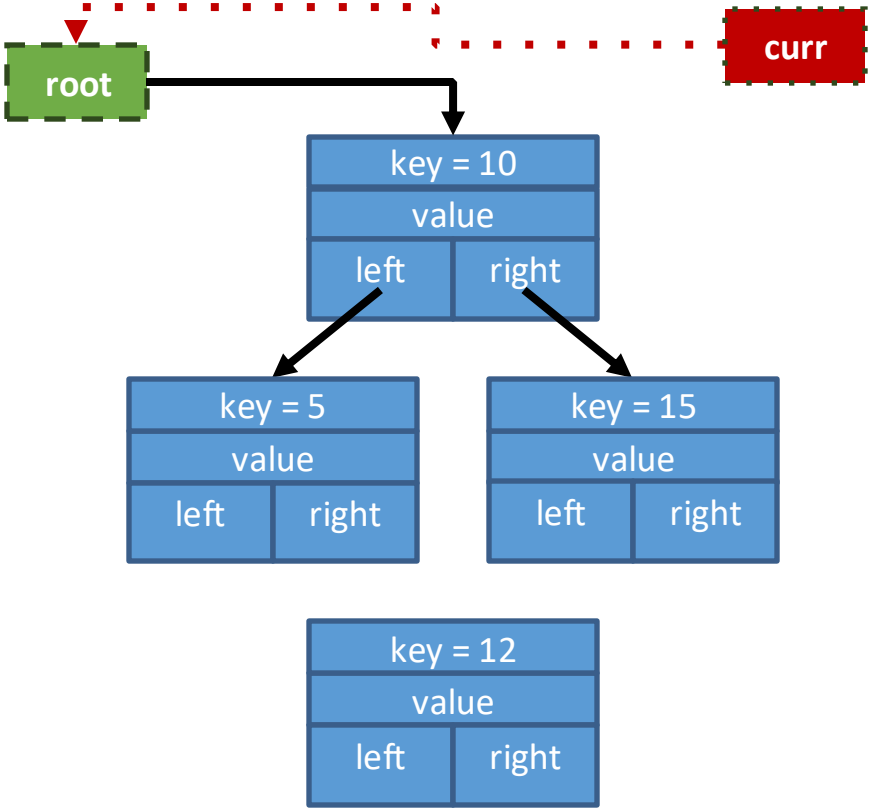
Task 6: Insert – another solution



Adapt the signature of the function!

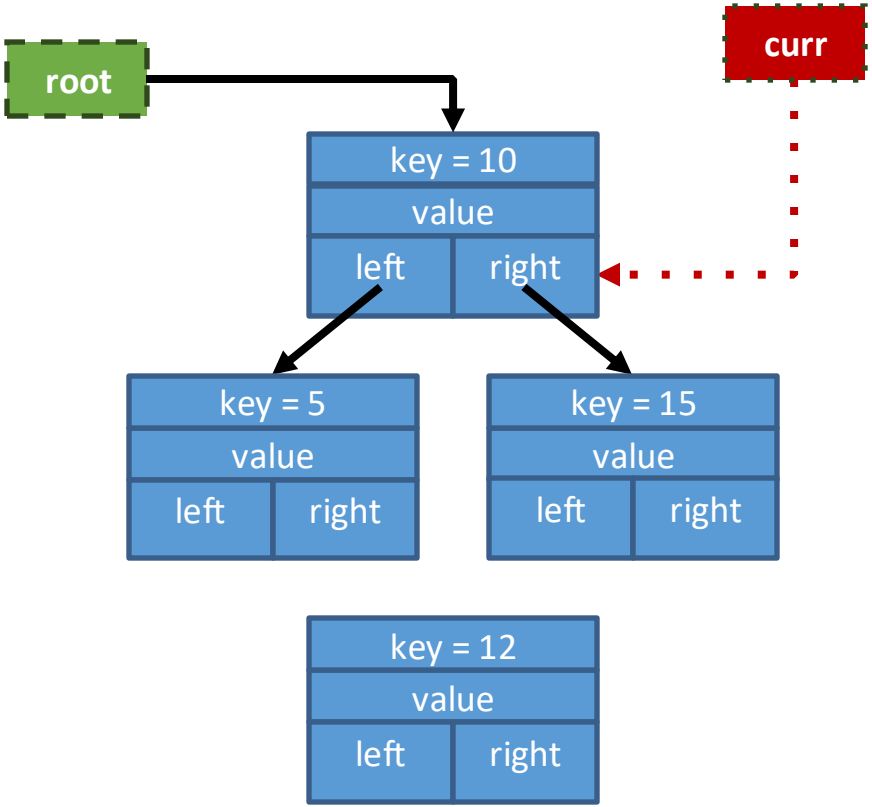
```
void insert(tree_node **curr, int key, int value) {
    if(*curr == NULL) {
                        *curr =
malloc(sizeof(tree_node));
                        (*curr)->key = key;
                        (*curr)->value = value;
                        return;
    }

    if ((*curr)->key > key)
            insert(&(*curr)->left, key, value);
    else if ((*curr)->key < key)
            insert(&(*curr)->right, key, value);
}


insert(&root, 12, 42);
```

Task 6: Insert – another solution



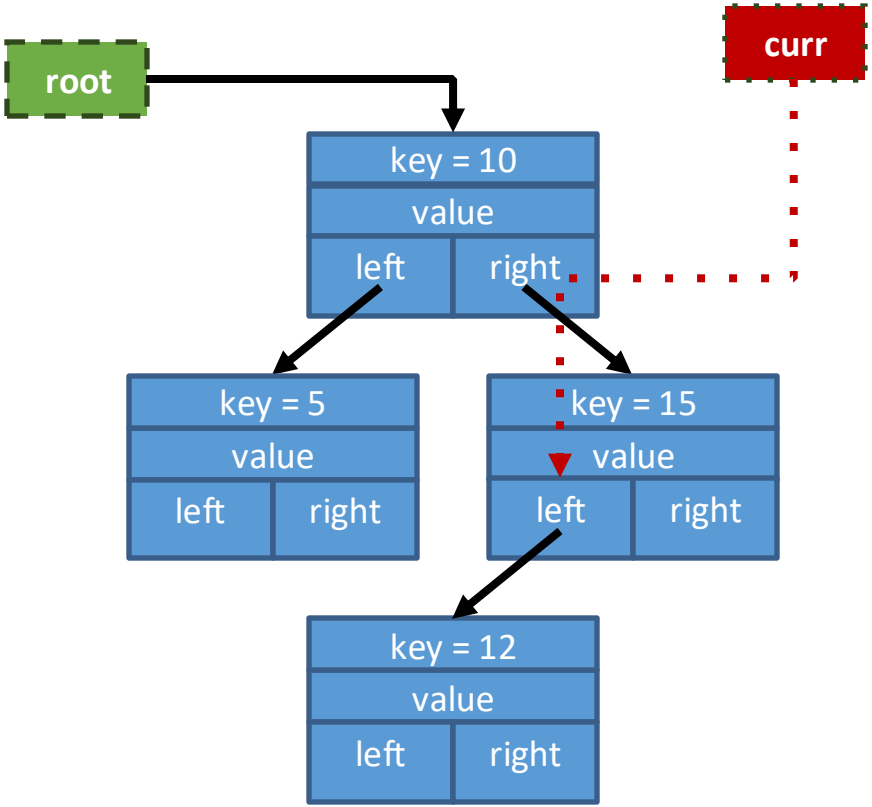

```
void insert(tree_node **curr, int key, int value) {
    if(*curr == NULL) {
                        *curr =
malloc(sizeof(tree_node));
                        (*curr)->key = key;
                        (*curr)->value = value;
                        return;
    }

    if ((*curr)->key > key)
            insert(&(*curr)->left, key, value);
    else if ((*curr)->key < key)
            insert(&(*curr)->right, key, value);
}


insert(&root, 12, 42);
```

Task 6: Insert – another solution



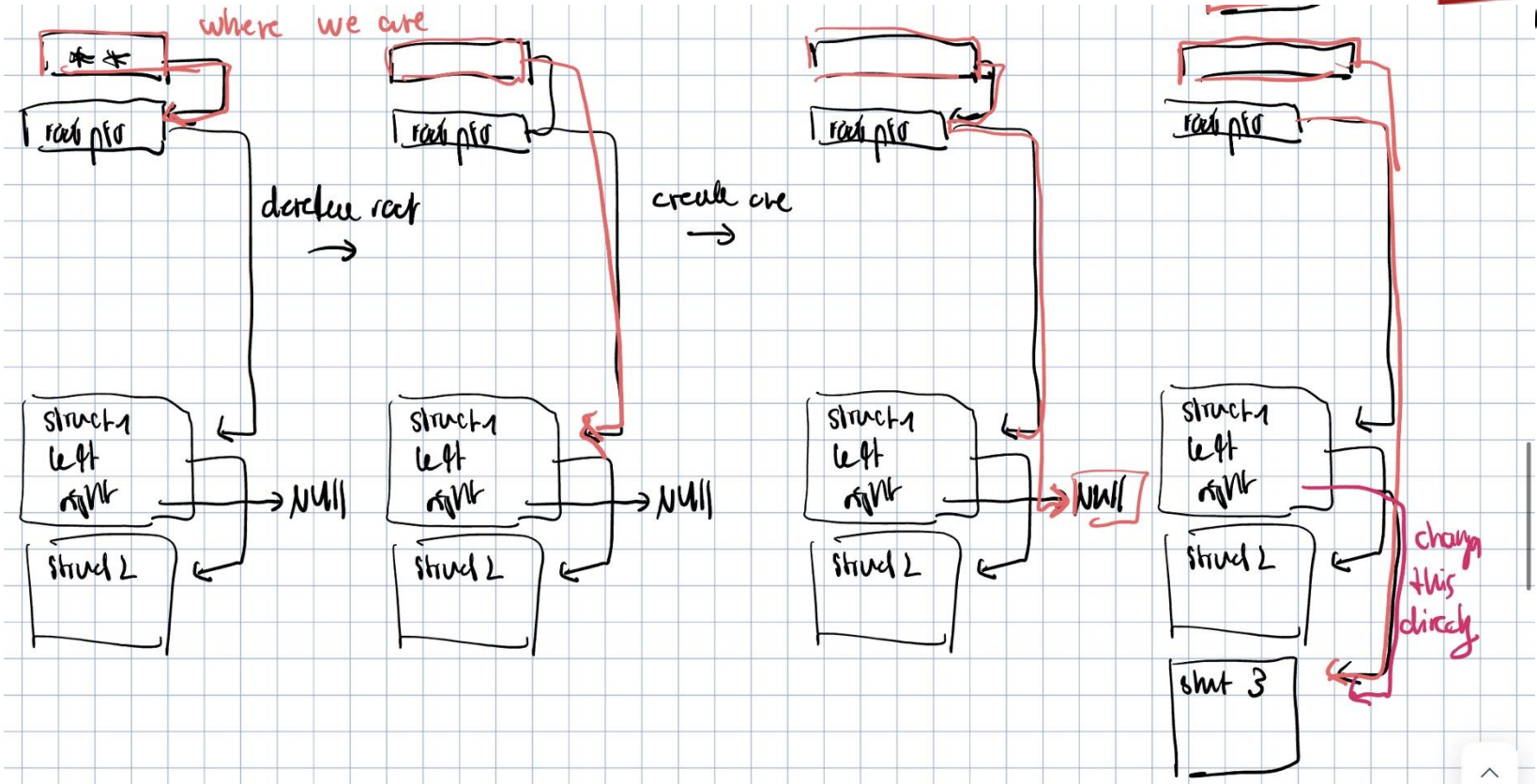

```
void insert(tree_node **curr, int key, int value) {
    if(*curr == NULL) {
                        *curr =
malloc(sizeof(tree_node));
                        (*curr)->key = key;
                        (*curr)->value = value;
                        return;
    }

    if ((*curr)->key > key)
            insert(&(*curr)->left, key, value);
    else if ((*curr)->key < key)
            insert(&(*curr)->right, key, value);
}


insert(&root, 12, 42);
```

Task 6: Insert – another solution




```
void insert(tree_node **curr, int key, int value) {
    if(*curr == NULL) {
                        *curr =
malloc(sizeof(tree_node));
                        (*curr)->key = key;
                        (*curr)->value = value;
                        return;
    }

    if ((*curr)->key > key)
            insert(&(*curr)->left, key, value);
    else if ((*curr)->key < key)
            insert(&(*curr)->right, key, value);
}


insert(&root, 12, 42);
```

Task 6 Double Pointer



Quiz

https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=1096740

Try to solve as many tasks as you can in **15 minutes**

Afterwards, we will discuss solutions

Quiz Solution a)

a) What does the following C code achieve?

```
p && *p
```

Quiz Solution a)

a) What does the following C code achieve?

```
p && *p
```

Solution: It prevents null pointer dereference.

- First, whether the pointer p is non-null
- Second, whether the value at *p is non-zero (truthy).

Quiz Solution b)

b) Consider the following program. What does it print out and why? (You may assume the call to malloc succeeds.)

```c
#include <malloc.h>
int main() {
  int a = 0;
  int *b = malloc(sizeof(int));
  if ((&a) > b) {
    printf("Trick!\n");
  } else {
    printf("Treat!\n");
  }
  return 0;
}
```

Quiz Solution b)

b) Consider the following program. What does it print out and why? (You may assume the call to malloc succeeds.)

```
#include <malloc.h>
int main() {
  int a = 0;
  int *b = malloc(sizeof(int));
  if ((&a) > b) {
    printf("Trick!\n");
  } else {
    printf("Treat!\n");
  }
  return 0;
}
```

Solution: Trick! Because the stack addresses are bigger than heap addresses. E.g. &a=0xbfc651d8, b=0x81e3008.

Quiz Solution b)

```c

int main(void) {
    int a = 0;
    int *b = malloc( size: sizeof(int));
    if((&a)>b){
        printf("Trick!\n");
    }else{
        printf("Treat!\n");
    }
    return 0;
```

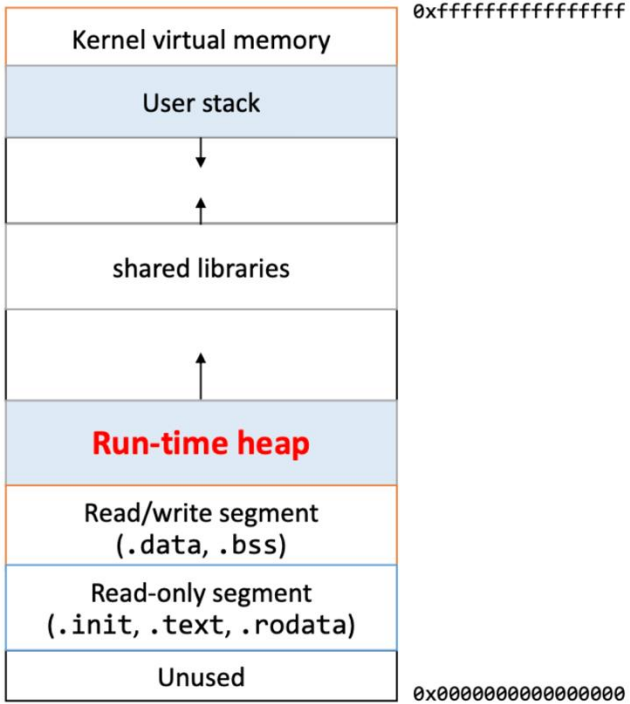
Run untitled8

```
/Users/benediktfalk/CLionProjects/untitled8/cmake-build-debug/untitled8
Trick!
```

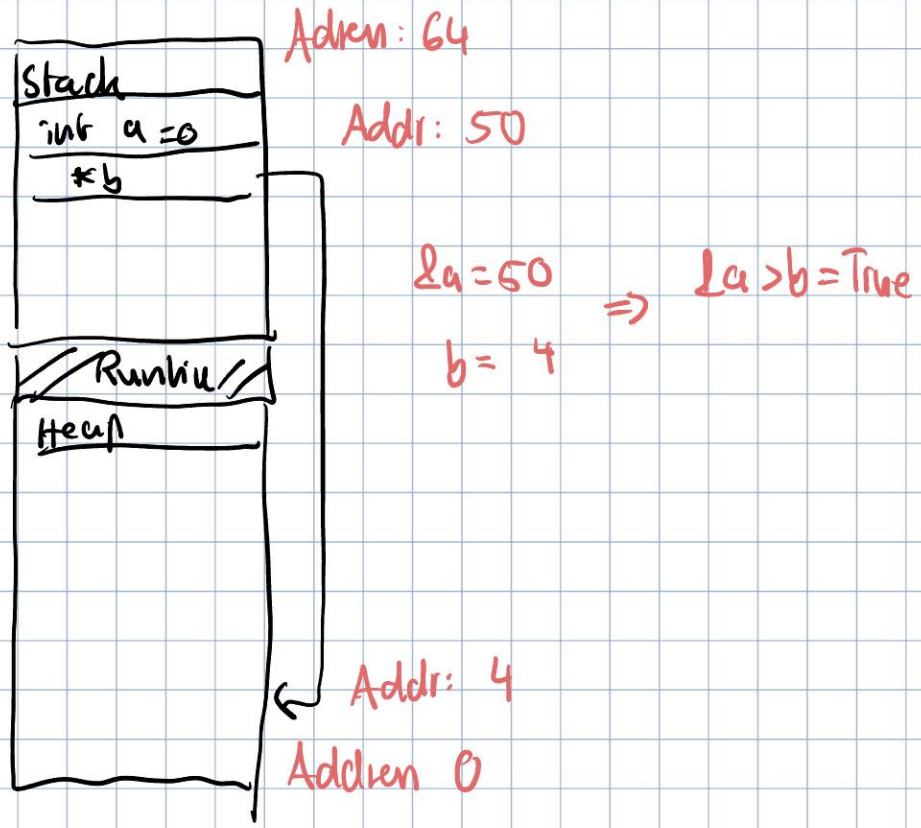
Quiz Solution b)

Where does all this memory come from?

- The heap (or “free store”)
  - Large pool of unused memory, used for dynamically allocated data structures
  - `malloc()` allocates chunks of memory in the heap, `free()` returns them
  - `malloc()` maintains bookkeeping data in the heap to track allocated blocks.



Quiz


Stack
int a = 0
*b
Runtime
Heap
Adren: 64
Addr: 50
&a = 50
b = 4
⇒
&a > b = True
Addr: 4
Addren 0

Q

```c
int main(void) {
    int a = 0;
    int *b = malloc( size: sizeof(int));
    printf("a: %p\n", &a);
    printf("b: %p\n", b);
    if((&a)>b){
        printf("Trick!\n");
    }else{
        printf("Treat!\n");
```

Run untitled8

```
/Users/benediktfalk/CLionProjects/untitled8/cmake-build-debug/untitled8
a: 0x7ff7bf708548
b: 0x600003230030
Trick!
```

Quiz Solution c)

c) What functionality is achieved by the function *foo*?

```
int foo(int b, int n) {
  int i, p;

  p = 1;
  for (i = 1; i <= n; ++i)
    p = p * b;
  return p;
}
```

Quiz Solution c)

c) What functionality is achieved by the function *foo*?

```
int foo(int b, int n)
{
   int i, p;

   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * b;
   return p;
}
```

Solution: The Code implements the function b^n.

Quiz Solution d)

d) What functionality is achieved by the function *foo*?

```
void foo(char *s, char *t) {
  while ((*s++ = *t++) != '\0')
    ;
}
```

Quiz Solution d)

d) What functionality is achieved by the function *foo*?

```
void foo(char *s, char *t)
{
  while ((*s++ = *t++) != '\0')
    ;
}
```

Solution: The Code implements the function `strcpy`.

Quiz Solution d)

```
5   void foo(char *s, char *t){
6       while(*t != '\0'){
7           *s=*t;
8           s++;
9           t++;
10      }
11  }
```

Quiz Solution e)

e) Strings and Pointers: What does the following program print?

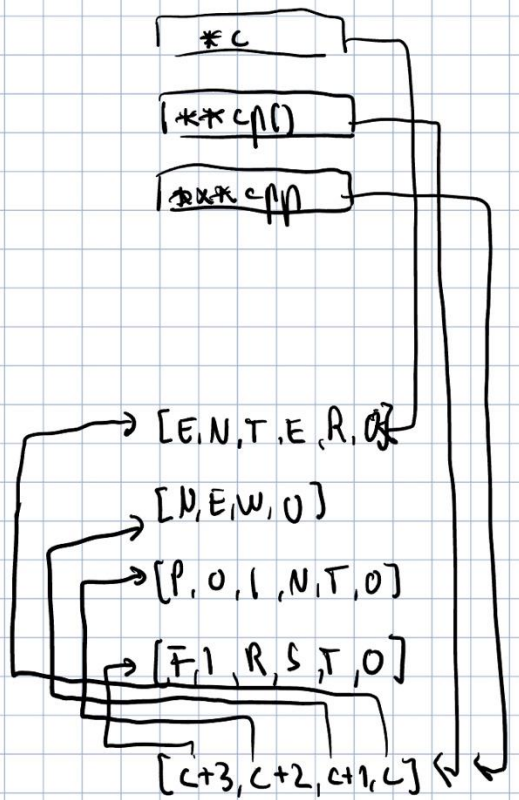
```
char *c[] = {
    "ENTER", "NEW", "POINT", "FIRST"
};
char **cp[] = { c+3, c+2, c+1, c };
char ***cpp = cp;
main() {
    printf("%s", **++cpp);
    printf("%s", *--*++cpp+3);
    printf("%s", *cpp[-2]+3);
    printf("%s\n", cpp[-1][-1]+1);
}
```

Quiz Solution e)

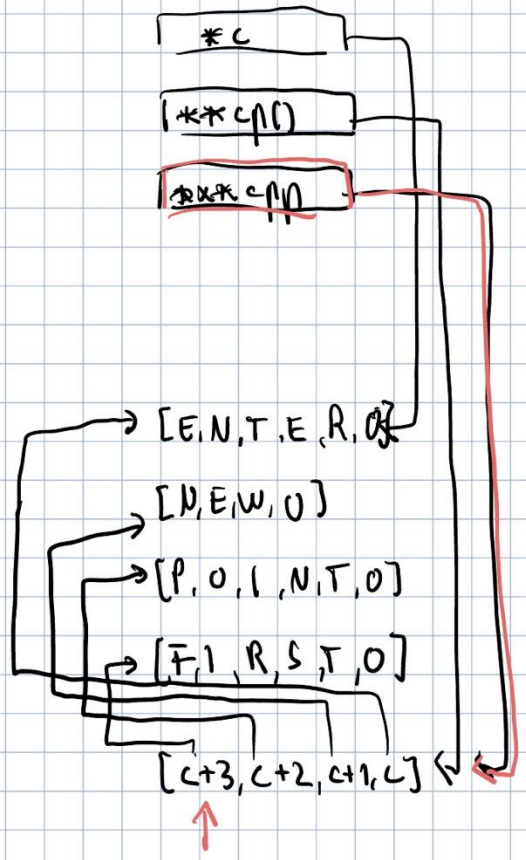
e) Strings and Pointers: What does the following program print?

```
char *c[] = {
   "ENTER", "NEW", "POINT", "FIRST"
};
char **cp[] = { c+3, c+2, c+1, c };
char ***cpp = cp;
main() {
   printf("%s", **++cpp);
   printf("%s", *--*++cpp+3);
   printf("%s", *cpp[-2]+3);
   printf("%s\n", cpp[-1][-1]+1);
}
```

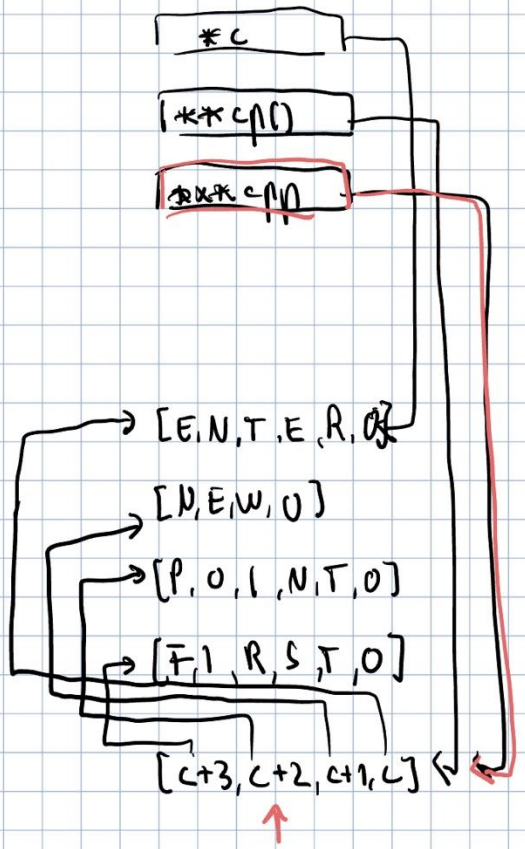
Solution: It prints POINTERSTEW.



**(++cpp)

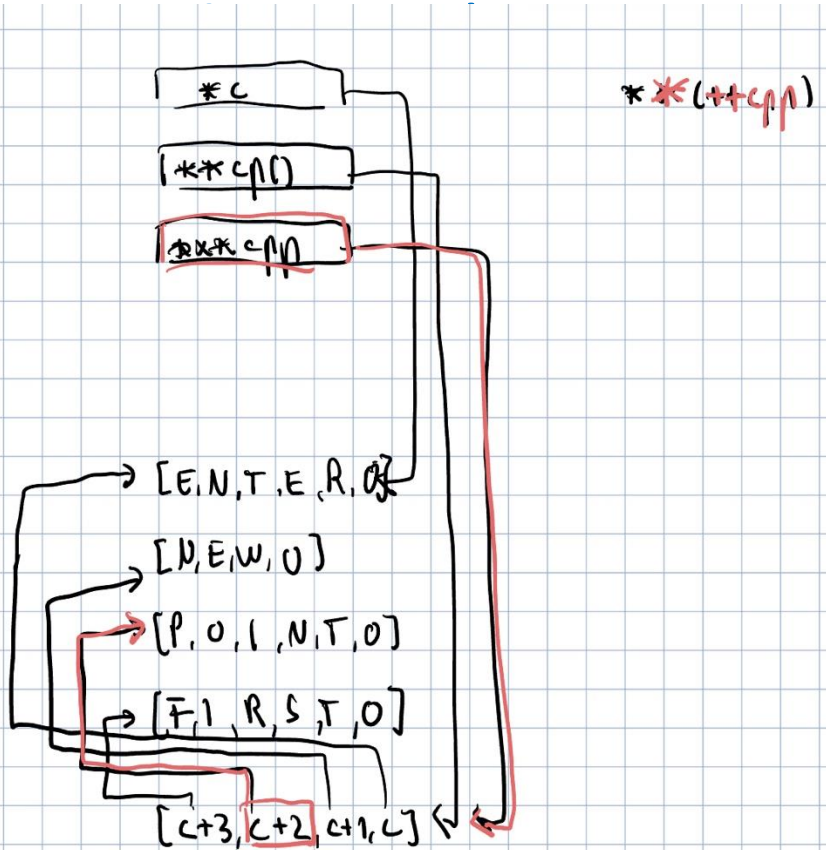


**++cpp

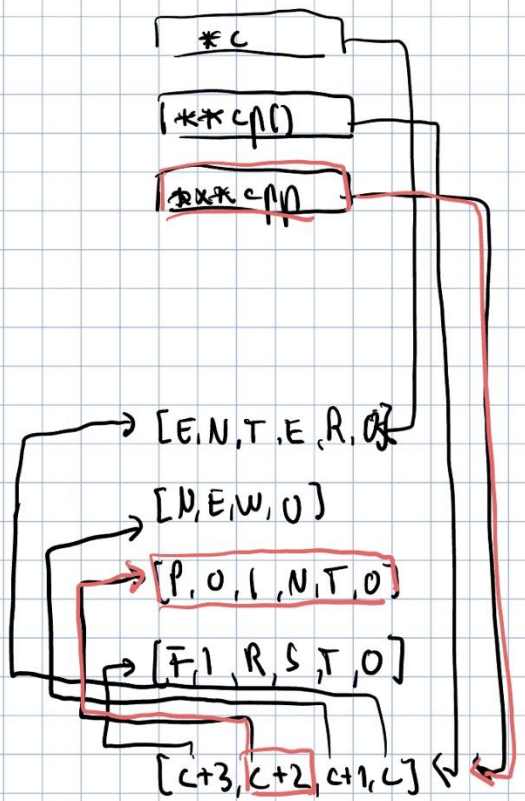


***(++cppp)

C



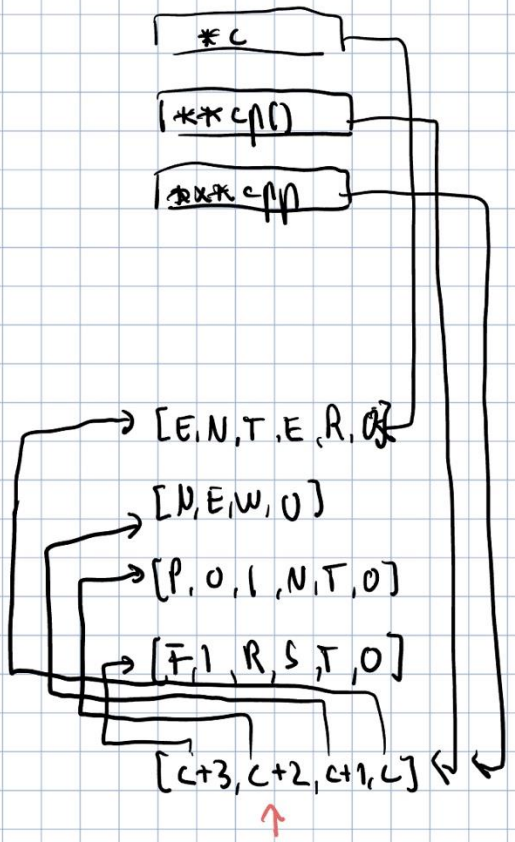
***(++cpp)



**(++cpp) =point

Quiz Solı

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

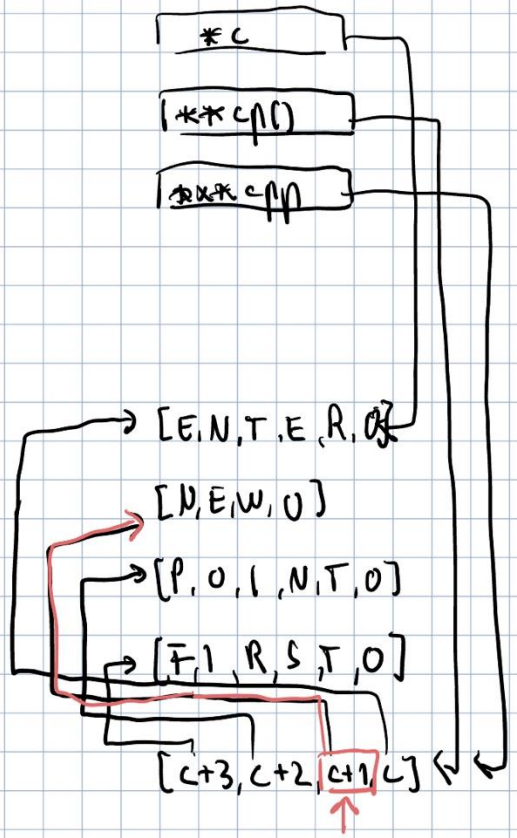


**(++cpp) = Point

((*(--(*(++cpp)))) +3)

Remember: cpp got incr.

Qu

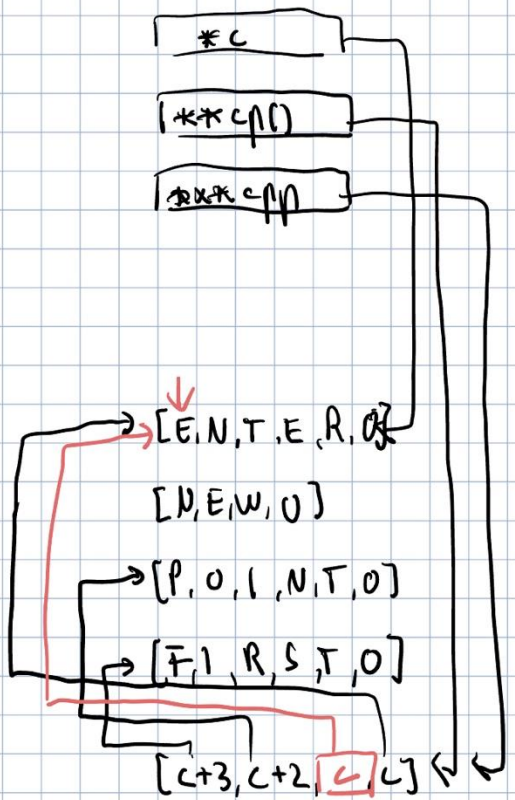


**(++cpp) = Point

((*(--(*(++cpp)))) +3)

Remember: cpp got incr.

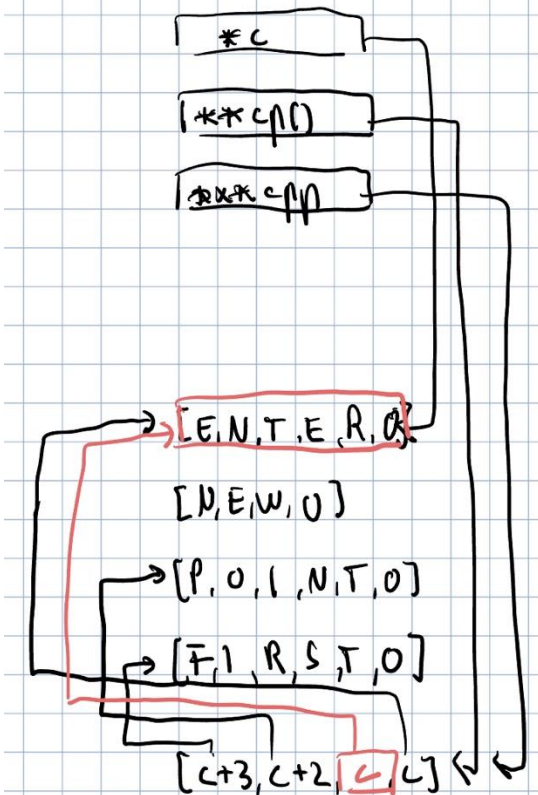
Qu



**(++cpp) = Point

((*(--(*(++cpp)))) +3)

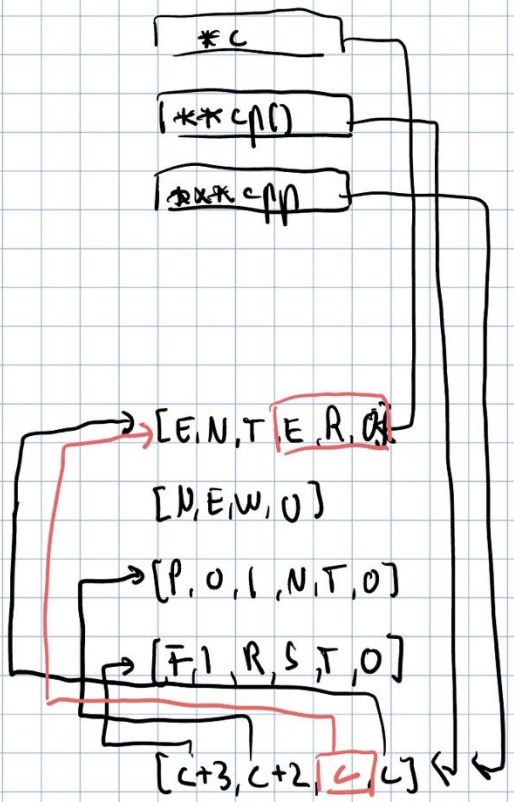
Remember: cpp got incr.



***(++cppp) = Point

((*(--(*(++cppp)))) +3)

Remember: cpp got incr.



**(++cpp) = Point

Remember: cpp got incr.

((*(--(*(++cpp)))) +3) = ER

Quiz Solution f)

f) Memory API: What does the following program print?

```
main() {
    int* ip = malloc(sizeof(int));
    printf("%d\n", *ip);
    int* ip2 = calloc(1, sizeof(int));
    printf("%d\n", *ip2);
    free(ip); free(ip2);
}
```

Quiz Solution f)

f) Memory API: What does the following program print?

```
main() {
    int* ip = malloc(sizeof(int));
    printf("%d\n", *ip);
    int* ip2 = calloc(1, sizeof(int));
    printf("%d\n", *ip2);
    free(ip); free(ip2);
}
```

Solution: It prints an undefined integer followed by 0.

Quiz Solution g)

g) Structs: What does the following program print?

```
#define PRINT(fmt, val) printf(#val " = %" #fmt "\t", (val))
#define PRINT2(fmt, v1, v2) PRINT(fmt, v1); PRINT(fmt, v2);

static struct S1 {
   char c[3], *s;
} s1 = {"abc", "def" };
static struct S2 {
   char *cp;
   struct S1 ss1;
} s2 = { "ghi", { "jkl", "mno" } };

PRINT2(c, s1.c[0], *s1.s);
PRINT2(s, s2.cp, s2.ss1.s);
PRINT2(s, ++s2.cp, ++s2.ss1.s);
```

Quiz S

g) Structs: What does the following program print?

```
#define PRINT(fmt, val) printf(#val " = %" #fmt "\t", (val))
#define PRINT2(fmt, v1, v2) PRINT(fmt, v1); PRINT(fmt, v2);

static struct S1 {
   char c[3], *s;
} s1 = {"abc", "def" };
static struct S2 {
   char *cp;
   struct S1 ss1;
} s2 = { "ghi", { "jkl", "mno" } };

PRINT2(c, s1.c[0], *s1.s);
PRINT2(s, s2.cp, s2.ss1.s);
PRINT2(s, ++s2.cp, ++s2.ss1.s);
```

Solution:

`PRINT2(c, s1.c[0], *s1.s)` prints

`s1.c[0] = a *s1.s = d`

`PRINT2(s, s2.cp, s2.ss1.s)` prints

`s2.cp = ghi s2.ss1.s = mno`

`PRINT2(s, ++s2.cp, ++s2.ss1.s)` prints

`++s2.cp = hi ++s2.ss1.s = no`

What are macros again?

- **Syntax**: #define NAME(parameters) expansion
- Name: Name of macro
- Parameters: List of arguments (if any)
- Expansion: Code that will replace the macro when used

- **Example**: #define SQUARE(x) (x * x)
- SQUARE(5) will be replaced by 5*5
- Macros get **expanded** (copy paste) by the preprocessor, its **substitution** NOT **evaluation**

What are macros again?

- Stringification with # ⇔ When using # infront of an argument it converts a macro into a string literal
- #define PRINT_VAR(x) printf(#x = “ = %d\n”, x)
- => PRINT_VAR(a) ⇔ printf(“a = %d\n”, a)

Direct structure initialization and naming

```
21  //Direct initialisation (i.e. can use directly)
22  static struct S3 {
23      char c[3];
24      char *s;
25  } s3 = { "abc", "def" };
26
27  int main(int argc, char** argv){
28    printf("%c\n", s3.c[0]);
29    return 0;
30  }
```

```
13  //Combine typedef and struct declaration into one
14  typedef struct S2 {
15      char c[3];
16      char *s;
17  } S2;
18
19  S2 s2 = { "abc", "def" };
```

Quiz S

g) Structs: What does the following program print?

```
#define PRINT(fmt, val) printf(#val " = %" #fmt "\t", (val))
#define PRINT2(fmt, v1, v2) PRINT(fmt, v1); PRINT(fmt, v2);

static struct S1 {
   char c[3], *s;
} s1 = {"abc", "def" };
static struct S2 {
   char *cp;
   struct S1 ss1;
} s2 = { "ghi", { "jkl", "mno" } };

PRINT2(c, s1.c[0], *s1.s);
PRINT2(s, s2.cp, s2.ss1.s);
PRINT2(s, ++s2.cp, ++s2.ss1.s);
```

Solution:

`PRINT2(c, s1.c[0], *s1.s)` prints

`s1.c[0] = a *s1.s = d`

`PRINT2(s, s2.cp, s2.ss1.s)` prints

`s2.cp = ghi s2.ss1.s = mno`

`PRINT2(s, ++s2.cp, ++s2.ss1.s)` prints

`++s2.cp = hi ++s2.ss1.s = no`

Quiz Solution g)

- This task is thus basically more a task about strings than anything else
- PRINT2(c, s1.c[0], *s1.s), translates to
- PRINT(c, s1.c[0]); // s1.c[0] is 'a'
- PRINT(c, *s1.s); // *s1.s dereferences the first character of "def", which is 'd'

Quiz S

g) Structs: What does the following program print?

```
#define PRINT(fmt, val) printf(#val " = %" #fmt "\t", (val))
#define PRINT2(fmt, v1, v2) PRINT(fmt, v1); PRINT(fmt, v2);

static struct S1 {
   char c[3], *s;
} s1 = {"abc", "def" };
static struct S2 {
   char *cp;
   struct S1 ss1;
} s2 = { "ghi", { "jkl", "mno" } };

PRINT2(c, s1.c[0], *s1.s);
PRINT2(s, s2.cp, s2.ss1.s);
PRINT2(s, ++s2.cp, ++s2.ss1.s);
```

Solution:

`PRINT2(c, s1.c[0], *s1.s)` prints

`s1.c[0] = a *s1.s = d`

`PRINT2(s, s2.cp, s2.ss1.s)` prints

`s2.cp = ghi s2.ss1.s = mno`

`PRINT2(s, ++s2.cp, ++s2.ss1.s)` prints

`++s2.cp = hi ++s2.ss1.s = no`

Quiz Solution g)

- PRINT2(s, s2.cp, s2.ss1.s); translates to
- PRINT(s, s2.cp); // s2.cp is "ghi"
- PRINT(s, s2.ss1.s); // s2.ss1.s is "mno”

- PRINT2(s, ++s2.cp, ++s2.ss1.s); translates to
- PRINT(s, ++s2.cp);   // Increment s2.cp to point to the second character of "ghi" -> "hi"
- PRINT(s, ++s2.ss1.s); // Increment s2.ss1.s to point to the second character of "mno" -> "no"

Assignment 3

malloclab

Remark Malloclab:

- I am not going to talk too much about malloclab => very extensively documented
- **Very good to learn C**: extensive C programming skills

malloclab

- Write your own **`malloc`**, **`realloc`** and **`free`**!
- Creatively explore the design space and implement ***an allocator*** that is *correct, efficient* and *fast*
- Evaluate your own implementation
  - The provided **`mdriver`** program will check the *throughput* and *utilization*
  - `mdriver` uses real and artificial application traces to evaluate your implementation
    - it replays allocation patterns (`malloc`, `realloc` and `free` calls) recorded from different applications

malloclab

- You will only modify the `mm.c` file in the handout
  - Implement the following functions: **`mm_init, mm_malloc, mm_free, mm_realloc`**
  - Feel free to define helper functions, variables etc.
- **Advice:**
  - Do your implementation in stages – First implement `malloc` and `free`, then start working on `realloc`
  - Your textbook contains a simple malloc reference implementation – Read and fully understand it first!
  - Start early – This is by far the most difficult and most sophisticated C code you wrote so far!

Remark Malloclab:

Systems@ETH Zürich

- I am not going to talk too much about malloclab => very extensively documented
- **Very good to learn C**: extensive C programming skills

Lecture Recap

Lecture Recap

Recall the malloc package

```
#include <stdlib.h>
```

`void *malloc(size_t size)`

Successful:

Returns a pointer to a memory block of at least `size` bytes (typically) aligned to 8- or 16-byte boundary

If `size == 0`, returns NULL

Unsuccessful: returns NULL (0) and sets `errno`

`void free(void *p)`

Returns the block pointed at by p to pool of available memory

p must come from a previous call to `malloc()` or `realloc()`

`void *realloc(void *p, size_t size)`

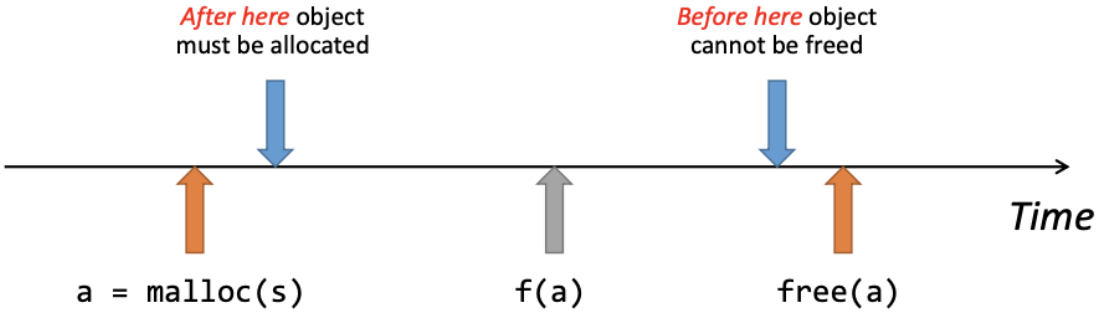
Changes size of block p and returns pointer to new block

Contents of new block unchanged up to min of old and new size

Old block has been `free()`'d (logically, if new != old)

Lecture Recap

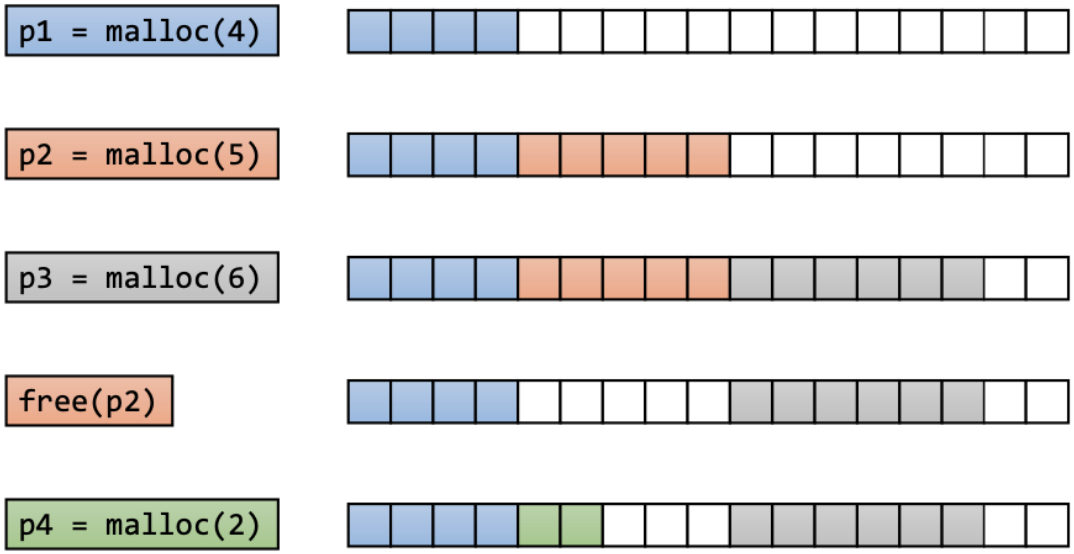
Explicit allocation



- E.g C, C++
- All operations appear in the program source

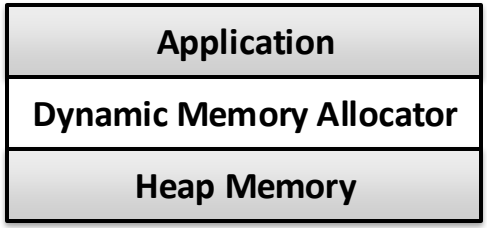
Lecture Recap

Allocation example



Dynamic Memory Allocation

- What is a memory allocator?
  - System software allocates pages of memory
  - An application typically uses memory in smaller pieces
  - The allocator's job is to manage the application's objects within the memory pages
- Allocation
  - Allowing an application to **allocate memory** means allowing it to ask for the memory it needs and then handing it memory blocks accordingly
  - A **memory block** is a contiguous range of bytes



An example of memory allocation

Where does everything go?

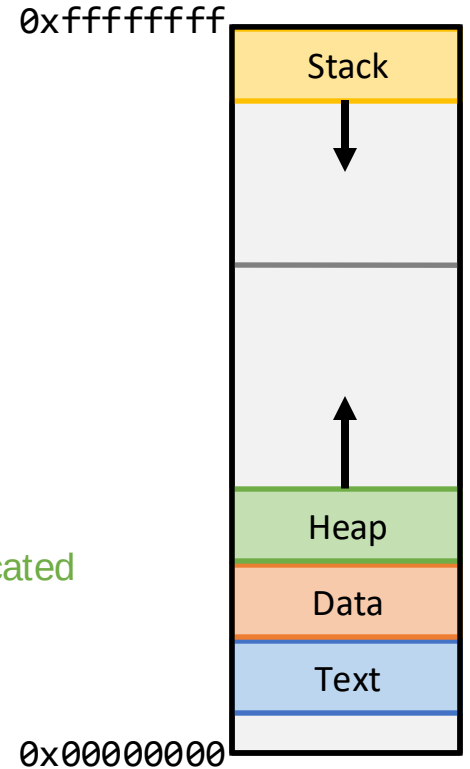
```
char big_array[1<<24]={1};  /* 16 MB  */
char huge_array[1<<28]={1}; /* 256 MB */
```

```
int beyond;
char *p1, *p2, *p3, *p4;
```

```
int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
}
```

Initialized data (.data)
Data
Uninitialized data (.bss)
Data
Program code (.text)
Text

Dynamically allocated memory
Heap



Explicit vs. implicit memory allocation

Systems@ETH Zürich

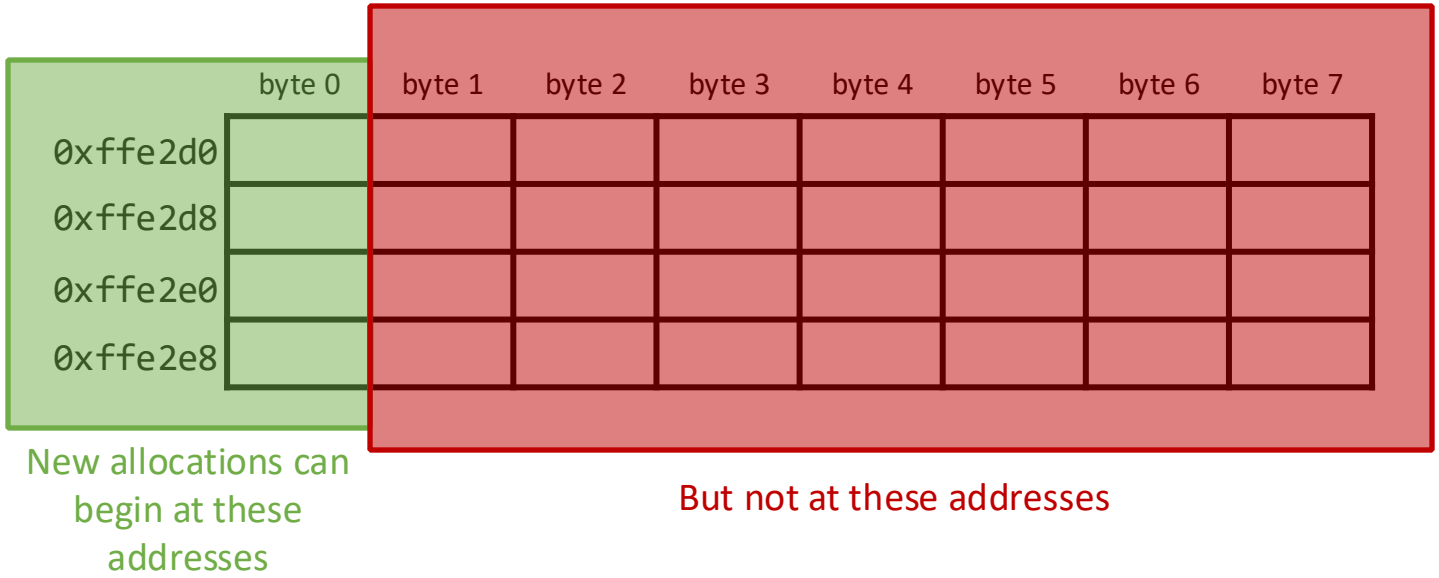
- In C, **explicit** memory allocation is used
  - The application allocates and frees space itself – `malloc()` and `free()`
  - C++ uses a similar approach – memory is handled **explicitly**
- In some other programming languages, the application must allocate memory but doesn't free it – **implicit** allocation
  - Java, ML, Lisp, C# etc.
  - Freeing memory is the job of a **garbage collector**

Allocator constraints

- An **application**…
  - Can issue an arbitrary sequence of `malloc()` and `free()` requests
  - Must issue `free()` only for blocks previously allocated using `malloc()`
- An **allocator**…
  - Can't control the number or size of blocks the application wishes to allocate
  - Must respond immediately to `malloc()` requests – can't reorder or buffer requests
  - Can only place new allocated blocks in free memory – no overlapping
  - Can manipulate and modify only free memory
  - Can't move around allocated blocks
  - Must follow alignment rules – 8-byte alignment

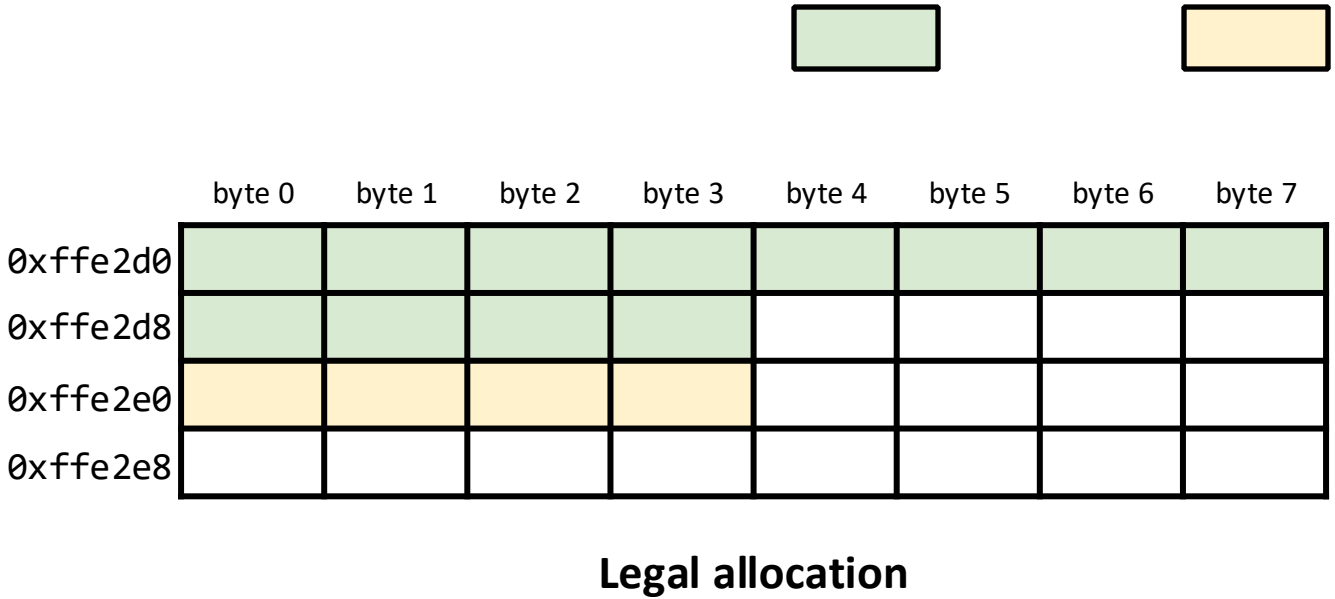
Alignment

- For consistency with the `libc malloc` package, your allocator must always return pointers that are **aligned to 8-byte boundaries**



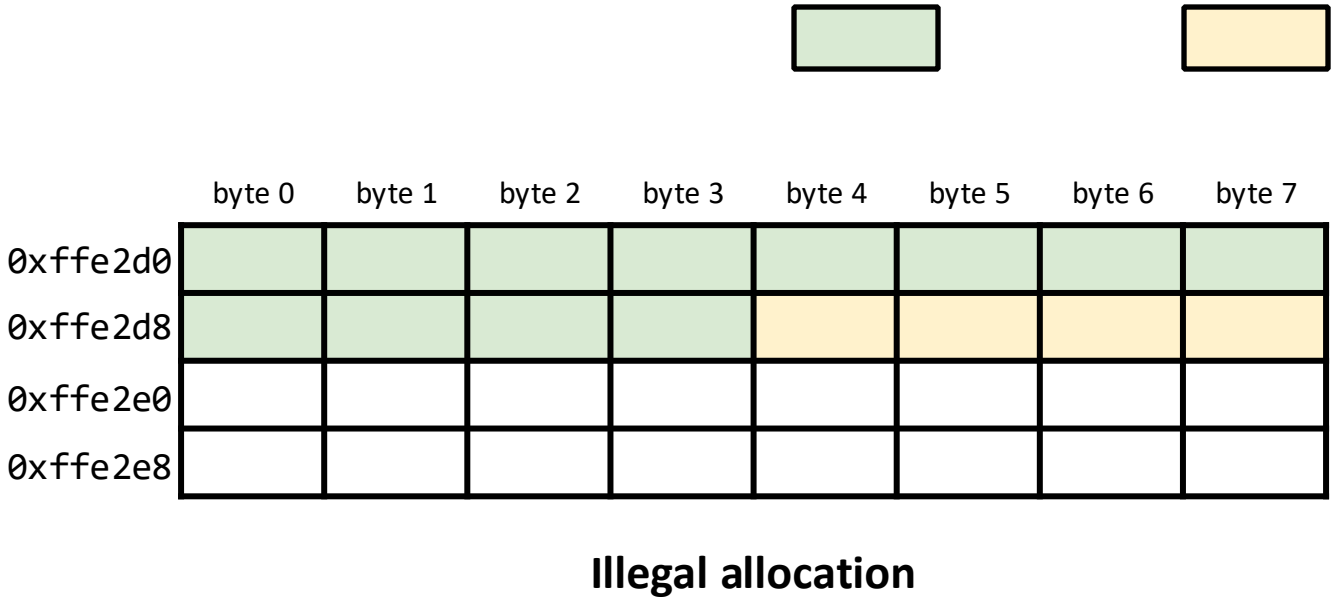
Alignment – example

- Consider 2 consecutive malloc() calls – one for 12 bytes and one for 4 bytes



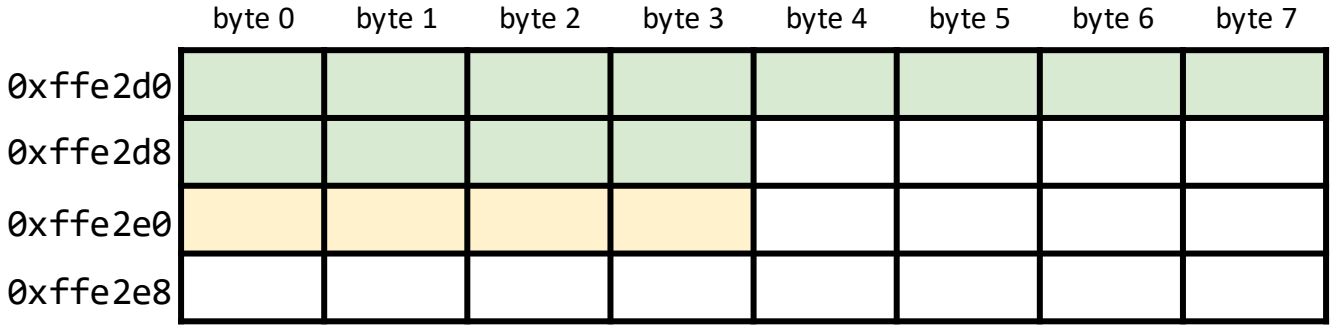
Alignment – example

- Consider 2 consecutive malloc() calls – one for 12 bytes and one for 4 bytes



Alignment

- Each new allocation must start at an address divisible by 8



Performance goals

Systems@ETH Zürich

- Your solution should have high *throughput* and *peak memory utilization*
  - These goals are often conflicting

- **Throughput**
  - Number of completed requests per unit of time
  - For an allocator that can handle 5000 `malloc()` and 5000 `free()` calls in 10 seconds, the throughput is 1000 operations per second

Performance goals

- **Peak memory utilization**
  - Your allocator can call a support routine **`void *mem_sbrk(int incr)`**
    - Expands the heap by `incr` bytes and returns a generic pointer to the first byte of the newly allocated heap area
  - The current **heap size** H_k is a monotonically non-decreasing value
    - It grows when `mem_sbrk` is called

Performance goals

- **Peak memory utilization**
  - Let's observe a sequence of n malloc() and free() requests

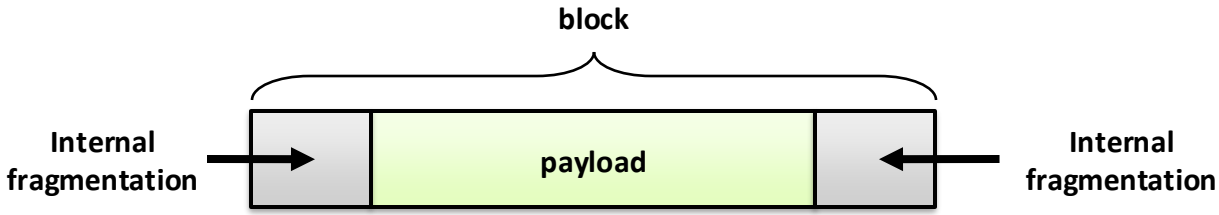
$$R_0, R_1, ..., R_k, ..., R_{n-1}$$

    - If R_k is a malloc(p bytes) request, it results in **a block with a payload of *p* bytes**
    - If R_k is a free() request for a block with a payload of *p* bytes, the *p* bytes of memory will be freed
  - **Aggregate Payload** P_k is calculated after request R_k has completed
    - P_k represents the sum of payloads of all malloc() requests amongst $R_0, ..., R_k$ minus the sum of sizes of all freed memory in requests $R_0, ..., R_k$
  - **Peak memory utilization** after k requests is then

$$U_k = \frac{max_{i \leq k} P_i}{H_k}$$

Internal fragmentation

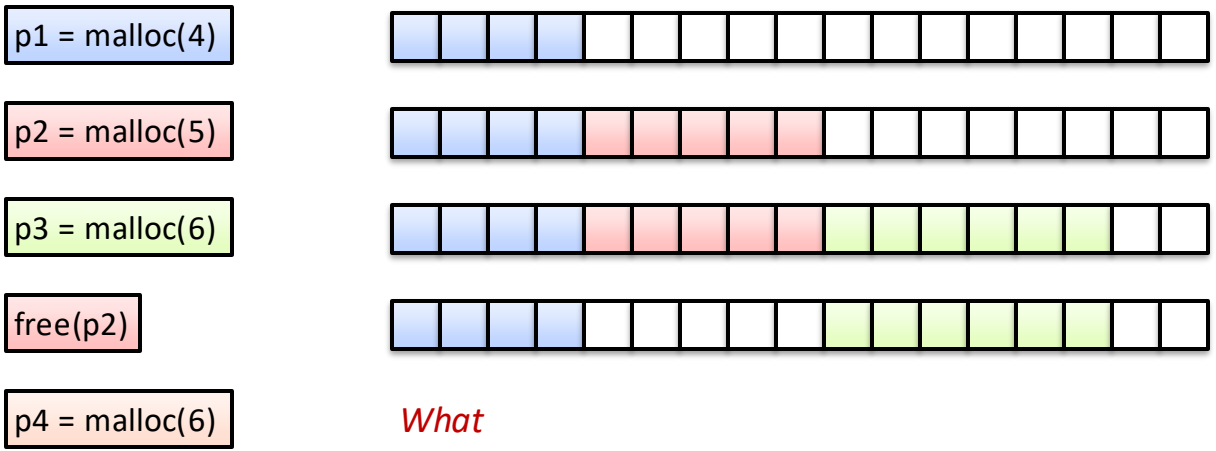
- For a given block, internal fragmentation occurs if **payload < block size**



- Caused by:
  - The overhead of maintaining the heap data structures
  - Padding for alignment purposes

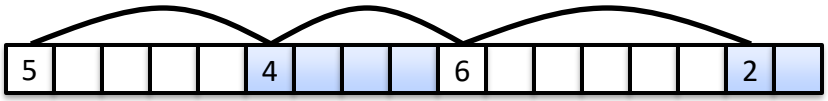
External fragmentation

- Occurs when there is enough aggregate heap memory, but **no single free block is large enough** to satisfy the current request

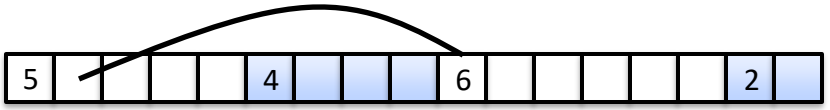


Keeping track of free blocks

1. **Implicit list –** using the length of blocks to implicitly link **all** blocks



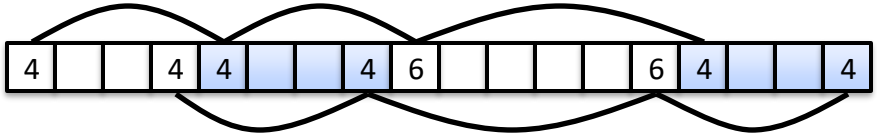
2. **Explicit list –** using pointers to link the free blocks



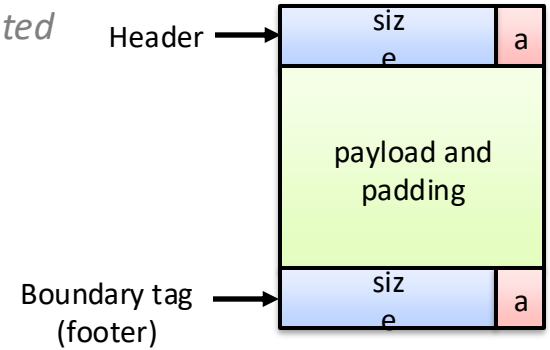
3. **Segregated free list –** keeping separate lists for free blocks of different sizes

Implicit list – bidirectional coalescing

- **Boundary tags** [Knuth73]
  - Replicate size/allocated word at the end of blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - An important and general technique!



The format of allocated and free blocks



a = 1: allocated block
a = 0: free block

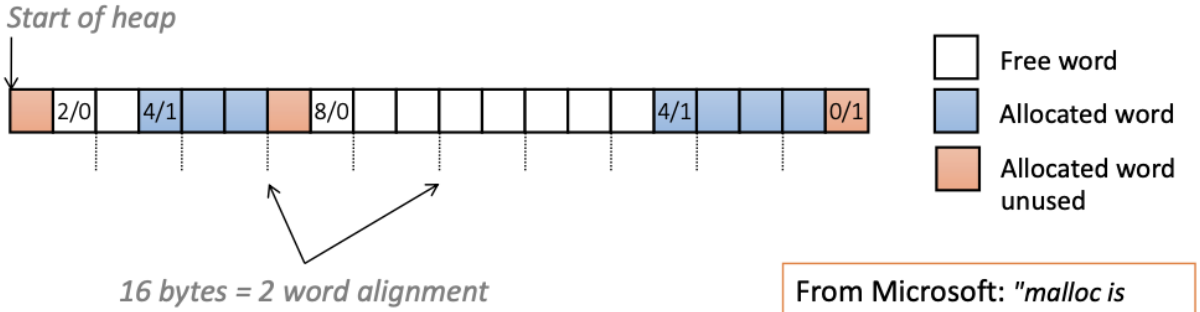
size: total block size

payload: application data
(allocated blocks only)

Implicit list

Example

Sequence of blocks in heap: 2/0, 4/1, 8/0, 4/1



- 16-byte alignment (x86_64)
  - May require initial unused word
  - Causes some internal fragmentation
- One word (0/1) to mark end of list
- Here: block size in words for simplicity

From Microsoft: *"malloc is guaranteed to return memory that's suitably aligned for storing any object that has a fundamental alignment and that could fit in the amount of memory that's allocated."*

Implicit list: policies

Systems@ETH Zürich

Implicit lists: Finding a free block

- First fit:
  - Search list from beginning, choose first free block that fits: (Cost?)

```
p = start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p <= len))) {   \\ too small
    p = p + (*p & -2);   \\ goto next block (word addressed)
}
```

  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list

Implicit list: policies

Systems@ETH Zürich

- Next fit:
  - Like first-fit, but search list starting where previous search finished
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first-fit

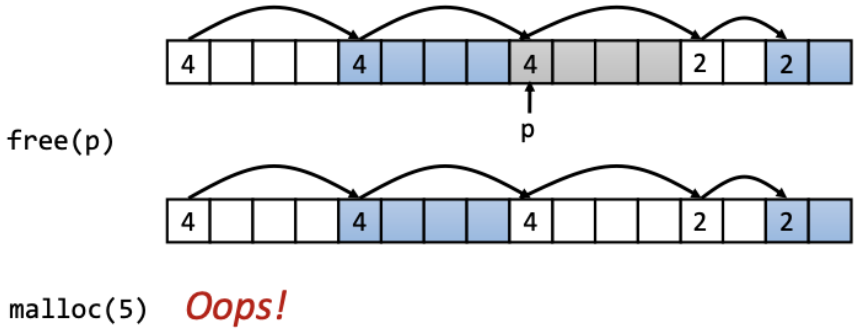
Coalescing

Implicit list: freeing a block

- Simplest implementation:
  - Need only clear the “allocated” flag

```
void free_block(ptr p) { *p = *p & -2 }
```

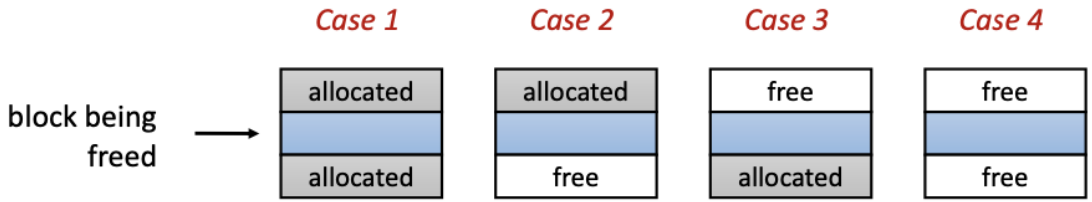
  - But can lead to “false fragmentation”



There is enough free space, but the allocator won’t be able to find it

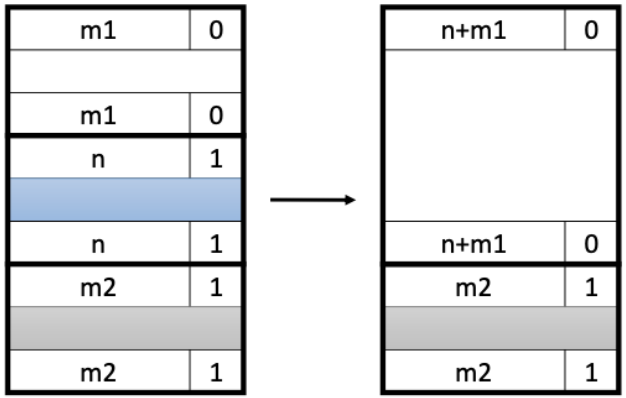
Coalescing

Constant time coalescing



Coalescing in implicit free lists

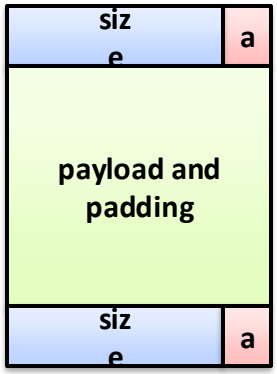
Constant time coalescing: case 3



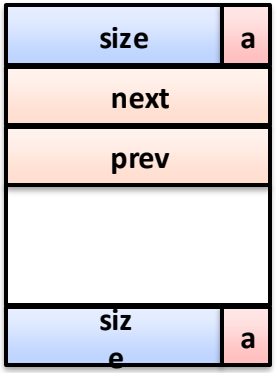
Explicit free lists

- Maintain list(s) of **free blocks**, not all blocks
- The “next” free block could be anywhere
  - We need to store forward/back pointers, not just sizes
  - Luckily, we link only free blocks, so we can use payload area
- Still need boundary tags for coalescing

Allocated (as before)



Free

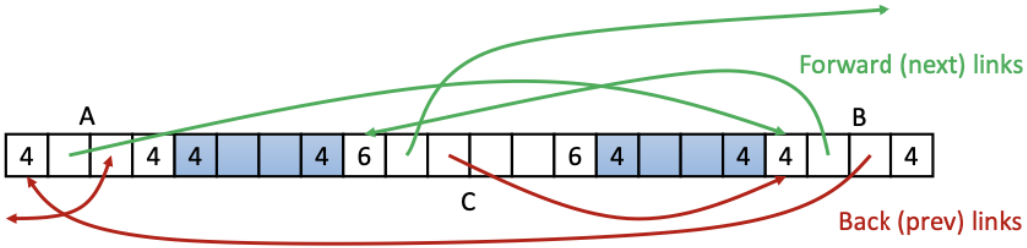


Explicit Free lists

Explicit free lists

- Logically:

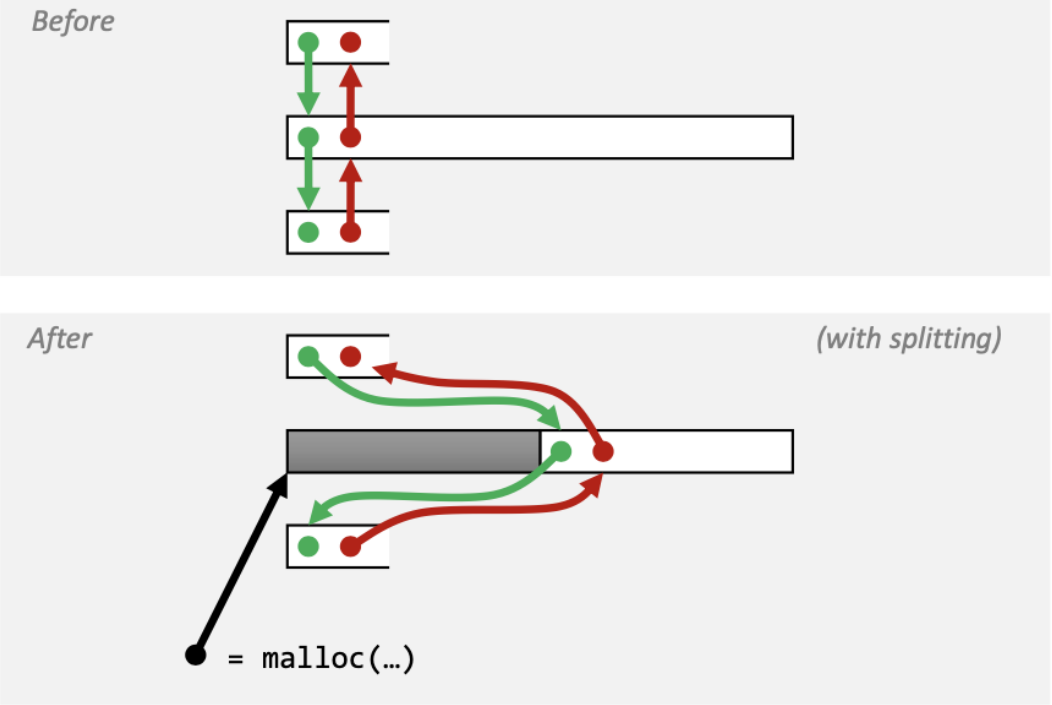
- Physically: blocks can be in any order



Explicit Free lists

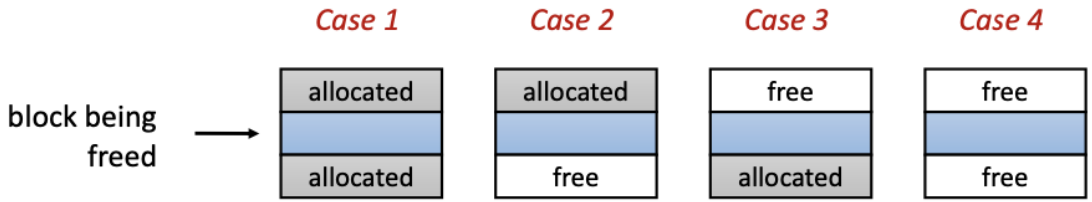
Allocating from explicit free lists

conceptual graphic



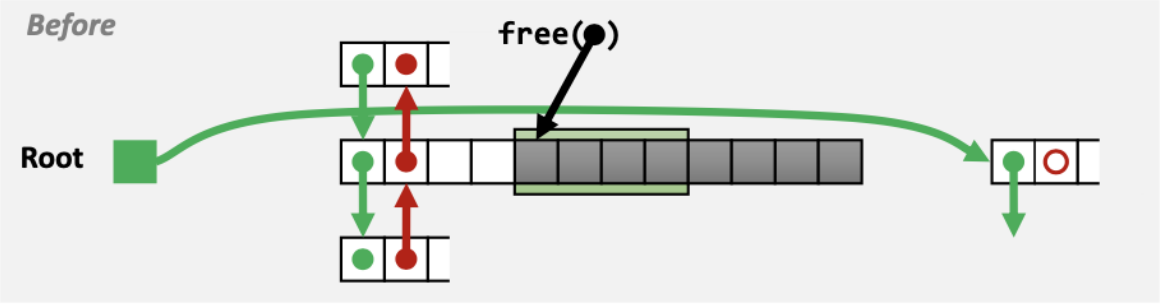
Coalescing Generally

Constant time coalescing

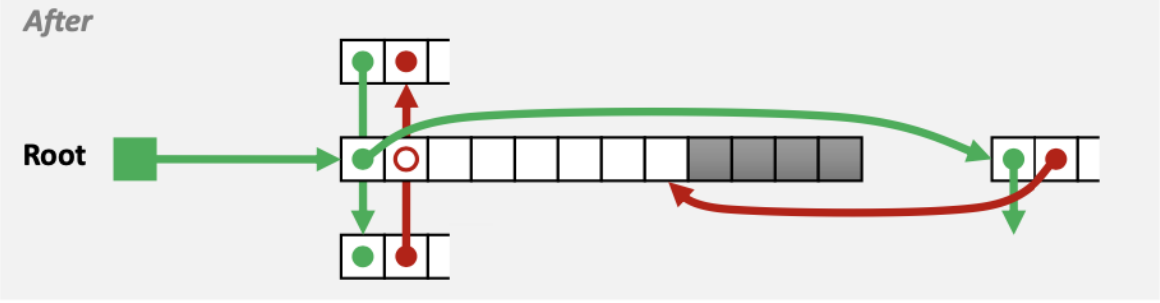


Coalesing in explicit free lists

Freeing with LIFO policy: case 2



- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



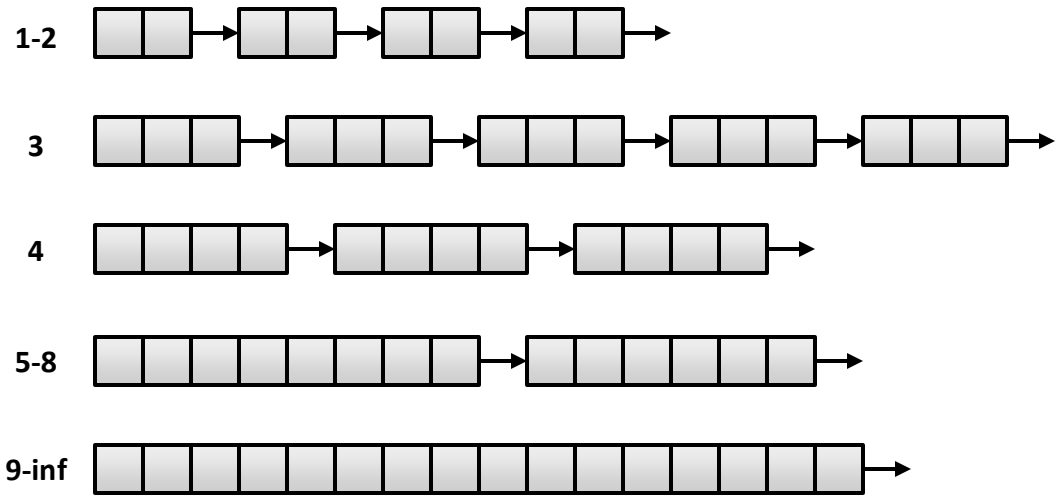
Explicit free lists

- **Insertion policy –** Where in the free list do you put a newly freed block?
  - **LIFO (Last In First Out) policy**
    - Insert the new block at the beginning of the free list
    - *Pro:* simple solution, constant time
    - *Con:* studies suggest high fragmentation
  - **Address-ordered policy**
    - Insert new blocks such that the free list always holds blocks in address order

      $addr_{prev} < addr_{curr} < addr_{next}$
    - *Pro:* studies suggest lower fragmentation than **LIFO**
    - *Con:* inserting a block requires searching

Segregated free lists

- Each **size class** of blocks has its own free list
  - A separate class is often kept for each small size
  - Whereas for larger sizes, one class is kept for each two-power size



Segregated Free lists

Seglist allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size *n*:
  - Search appropriate free list for block of size *m* > *n*
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

Key allocator policies

- **Placement policy**
  - *First-fit, next-fit, best-fit*, etc.
  - Trade off between throughput and fragmentation
  - Interesting observation – **segregated free lists** approximate a best fit placement policy without having to search entire free list
- **Splitting policy**
  - When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- **Coalescing policy**
  - ***Immediate coalescing*** **–** coalesce each time `free()` is called
  - ***Deferred coalescing*** **–** try to improve the performance of `free()` by deferring coalescing until needed
    - Coalesce as you scan the free list for `malloc()`
    - Coalesce when the amount of external fragmentation reaches some threshold

Old exam questions regarding lists

Systems@ETH Zürich

- HS10 Question 9
- HS11 Question 13
- HS12 Question 13

- Has not really appeared in recent exams: note that your exam is however different (you may need to implement the code for an explicit free list w/ coalescing or the like)

Remark: Memory pitfalls

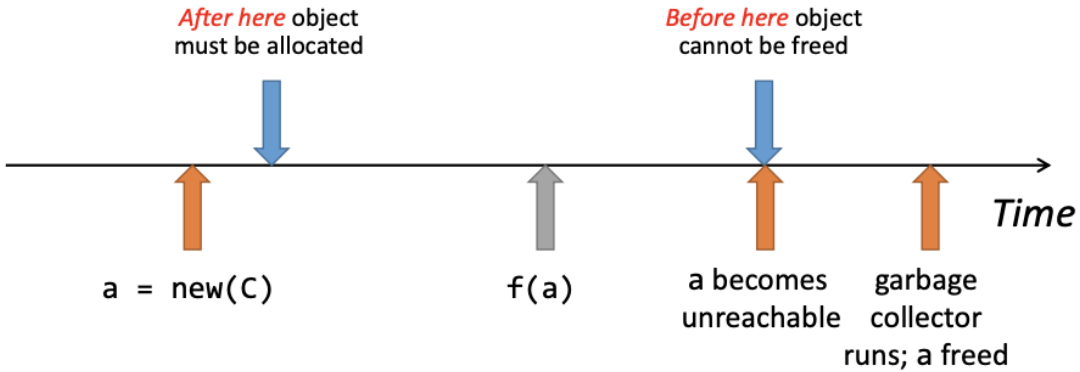
Systems@ETH Zürich

Memory-related perils and pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Remark: Garbage Collection (Java)

Garbage-collected

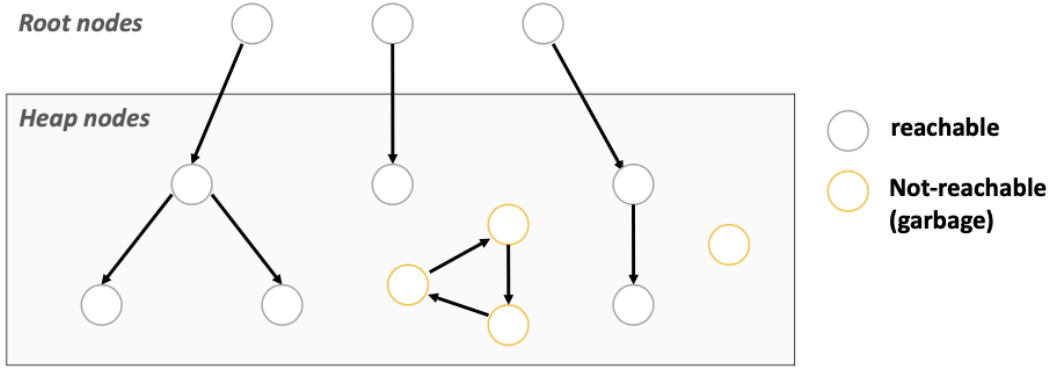


- E.g. Java, ML, Lisp, Python
- Explicit allocation, implicit deallocation

Remark: Garbage Collection (Java)

Memory as a graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)



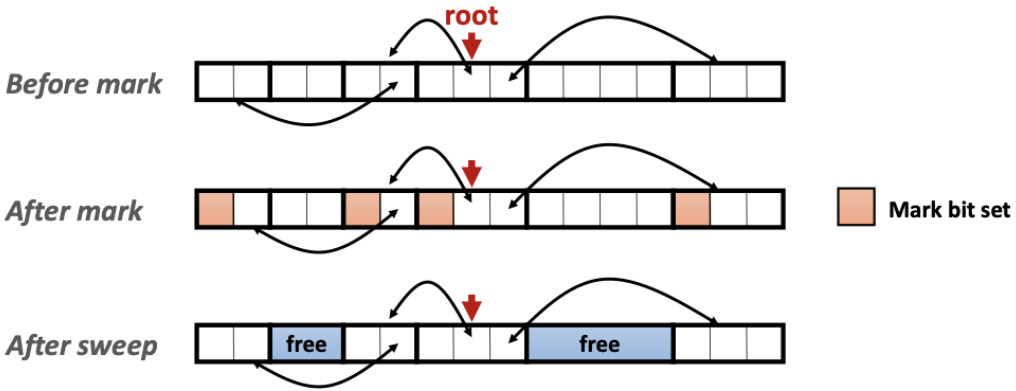
A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (cannot be needed by the application)

Remark: Garbage Collection (Java)

Mark and Sweep collecting

- Can build on top of malloc/free package
  - Allocate using malloc until you "run out of space"
- When out of space:
  - Use extra ***mark bit*** in the head of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



Outlook SPCA

Systems@ETH Zürich

- Basically done with: I (new lecture on how to write test case)
- **I: Programming Language C** (C Integers, Pointers, Preprocessor, Dynamic Memory Allocation)
- **II: Assembly x86-64** (x86 Assembly, Compiling C Data Structures, Linking and Loading, Compilers)
- **III: Computer Architecture** (Architecture and Optimisation, Caches, Exceptions, Virtual Memory)

Good luck!

- Build a **heap consistency checker** to help you with debugging and coding your allocator (more details in the assignment sheet)
- Use the **`memlib`** package to interact with the (simulated) memory system (more details in the assignment sheet)
- Test and benchmark your code with **`mdriver`**
- Refer to **these slides** and to **Lecture 6** for a better understanding of dynamic memory allocation