

---

# COMP2017 / COMP9017      Week 13 Tutorial

---

Revision

## Question 1: Unit of Study Survey (USS)

Please take 5-10 minutes to fill out the Unit of Study Survey:

<https://student-surveys.sydney.edu.au/students/>

## Question 2: Binary Search Tree (Part 1)

Your task is to build an unbalanced binary search tree in C!

A binary search tree is a lot like a linked list. Each node has two links, a left link and a right link. Following the left link visits smaller values in the subtree, and following the right link visits larger values. To maintain this ordering, values must be inserted in the correct place.

At a high-level, to insert, you traverse the tree, comparing the value in the current node with the value to insert. If the value to insert is smaller, traverse left, if the value to insert is larger, traverse right. Eventually, the spot to insert is found and the value is inserted.

Implement the unbalanced binary search tree below, ensuring that all dynamic memory is freed as soon as it is no longer needed.

```
typedef struct tree_node tree_node;
typedef struct bst bst;

struct tree_node
{
    int val;
    tree_node* left;
    tree_node* right;
};

struct bst
{
    tree_node* root;
};

/* Construct the binary search tree */
bst* init_bst()
```

```
{  
  
}  
  
/* Insert value 'val' into the binary search tree */  
void bst_insert(bst* root, int val)  
{  
  
}  
  
/* Return 1 if the target value in the binary search tree exists otherwise 0 */  
int bst_exists(bst* root, int target)  
{  
  
}  
  
/* Destroy the binary search tree */  
void destroy_bst(bst* root)  
{  
  
}
```

### Question 3: Binary Search Tree (Part 2)

The binary search tree implementation above can only hold **int** types. We now want to extend the BST to support generic types by holding a user supplied **void\***.

This introduces the following questions:

1. How do you compare **void\*** values inside the binary search tree? Is there a way that the user can tell the data structure how to compare them?
2. Who “owns” the memory for each element i.e. who is responsible for deallocating its memory?

### Question 4: Processes and Resource Management

What are the resource leaks in the following code?

What complexities need to be considered when managing resources in a multi-process system?

```
#define MAX_MSG_LEN 50  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <string.h>
```

```
#include<sys/wait.h>

void child_routine(int fd)
{
    // read messages from the parent and repeat them
    for (int i = 0; i < 5; ++i)
    {
        char msg[MAX_MSG_LEN] = {'\0'};
        read(fd, msg, MAX_MSG_LEN);
        printf("Echoing: %s\n", msg);
    }
}

void parent_routine(dyn_arr* child_fds)
{
    // for every child
    for (int j = 0; j < child_fds->size; ++j)
    {
        // send heartwarming messages through the pipe to the child
        char song[5][MAX_MSG_LEN] = { "Never", "gonna", "give", "you", "up" };
        for (int i = 0; i < 5; ++i)
        {
            write(child_fds->vals[j], song[i], MAX_MSG_LEN);
        }
    }
    destroy_dyn_arr(child_fds);
}

int main()
{
    dyn_arr* child_fds = init_dyn_arr();

    // create 10 child processes
    for (int i = 0; i < 10; ++i)
    {
        // create a pipe for communication
        int fd[2] = {0};
        pipe(fd);

        // fork
        pid_t pid = fork();
        if (pid < 0)
        {
            fprintf(stderr, "fork failed");
            return 1;
        }

        // child process
        if (pid == 0)
        {

```

```

        child_routine(fd[0]);
        return 0;
    }

    // parent process saves the fd for later communication
    else
    {
        insert_dyn_arr(child_fds, fd[1]);
    }

    // send messages from the parent to the child
    parent_routine(child_fds);
    return 0;
}

```

## Question 5: Memory, sizeof and pointers

1. Assume a 64 bit system where the **sizeof** all pointers are 8 bytes, and **sizeof (int)** is 4 bytes. The following questions refer to the code below:

```

void f(void) {
    struct int_array {
        int* data;
        int len;
    };

    struct int_array arr;
    struct int_array* array_ptr = &arr;
    struct int_array arrs[100];
}

```

- What are the initial values of `arr.data` and `arr.len`?
- What is **sizeof(struct int\_array)**?
- What is **sizeof(array\_ptr)**?
- What is **sizeof(array\_ptr->data)**?
- What is **sizeof(arrs)**?
- Suppose the address of `arrs[0]` is `0x1000`, what is the address of `arrs[50]`?
- Suppose the address of `arrs[50]` is `0x1500`, what is the address of `arrs[20]`?
- Suppose the address of `arrs[50]->len` is `0x1500`, what is the address of `arrs[50]->data`?

2. Assume a 64 bit system, where the **sizeof(int)** is 4 bytes, and **sizeof(float)** is also 4 bytes. The following questions refer to the code below:

```

void *g(int x) {
    union merged {
        int x;
        float y;
    };
    static int counter = 0;
    ++counter;
    union merged *ptr = malloc(sizeof(union merged));
    if (!ptr) {
        return NULL;
    }
    ptr->x = x;
    return ptr;
}

```

- (a) What is `sizeof(union merged)`?
- (b) Which variables are allocated on the stack, which variables are allocated on the heap and which variables are allocated in the static storage area?
- (c) What are the lifetimes of each of the variables listed above?

## Question 6: Control Flow

1. Trace the following program by hand. What does it output?

```

#include <stdio.h>
int main(void) {
    for (unsigned i = 5; i-- > 0;) {
        printf("%u\n", i);
        if (i) {
            for (int j = 0; j < i; ++j) {
                printf("%d\n", i + j);
            }
        }
    }
}

```

2. Why does the following code segfault when `x` is a `char*`?

```

void cpy2(char *dest, char *src, size_t n) {
    size_t i = 0;
    for (; i < n; ++i) {
        dest[i] = src[i];
    }
}

int main(void) {
    char x[] = "123456";
}

```

```
    cpy2(x + 1, x, 4);  
    puts(x);  
    return 0;  
}
```

## Question 7: Strings

Implement the following functions from `string.h`:

```
void memcpy(void* dest, const void* src, size_t len);  
void memmove(void* dest, const void* src, size_t len);  
int memcmp(const void* s1, const void* s2, size_t len);  
void strcpy(char* dest, const char* src);  
void strcat(char* dest, const char* src);  
char* strchr(const char* s, int c);  
char* strrchr(const char* s, int c);  
size_t strspn(const char* s, const char* accept);
```

## Question 8: Parallelism & Concurrency

1. Why is `x += 1` not an atomic operation?
2. What are the four conditions of a deadlock?
3. What is a critical section?
4. What is a live lock?
5. What is starvation in terms of multi-threading?
6. What does it mean if a resource is highly contended, why would this cause a performance issue?
7. What is false sharing?
8. How can a binary semaphore be used as a mutex?
9. The Senate Bus Problem from “The Little Book of Semaphores”:

“This problem was originally based on the Senate bus at Wellesley College. Riders come to a bus stop and wait for a bus. When the bus arrives, all the waiting riders invoke `boardBus`, but anyone who arrives while the bus is boarding has to wait for the next bus. The capacity of the bus is 50 people; if there are more than 50 people waiting, some will have to wait for the next bus. When all the waiting riders have boarded, the bus can invoke `depart`. If the bus arrives when there are no riders, it should depart immediately. Puzzle: Write synchronization code that enforces all of these constraints. ”

Represent each rider as a thread, write synchronisation code to model the above problem.