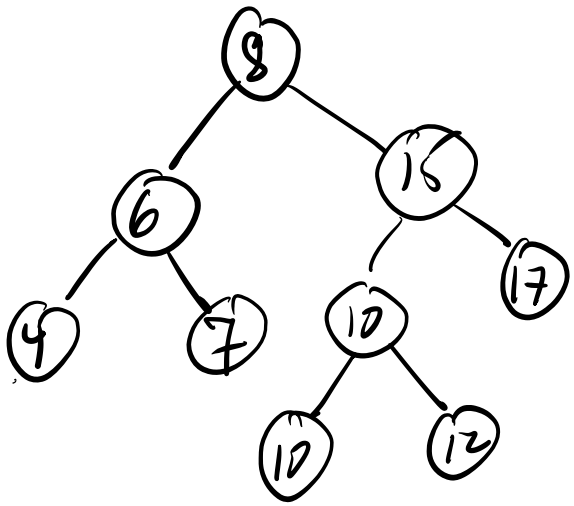
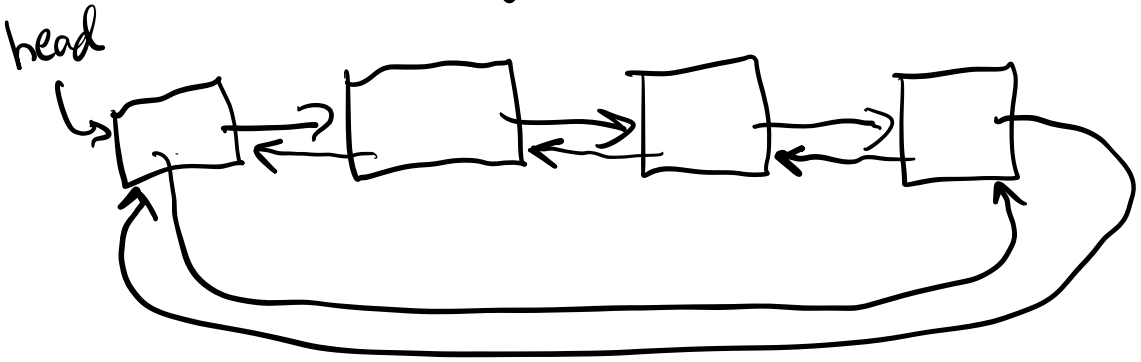
Programming Assignment 1

- Binary Search Trees
  - The value at the left child of a parent node is less than or equal to the value at the parent node.
  - The value at the right child is greater than the parent.

Example:

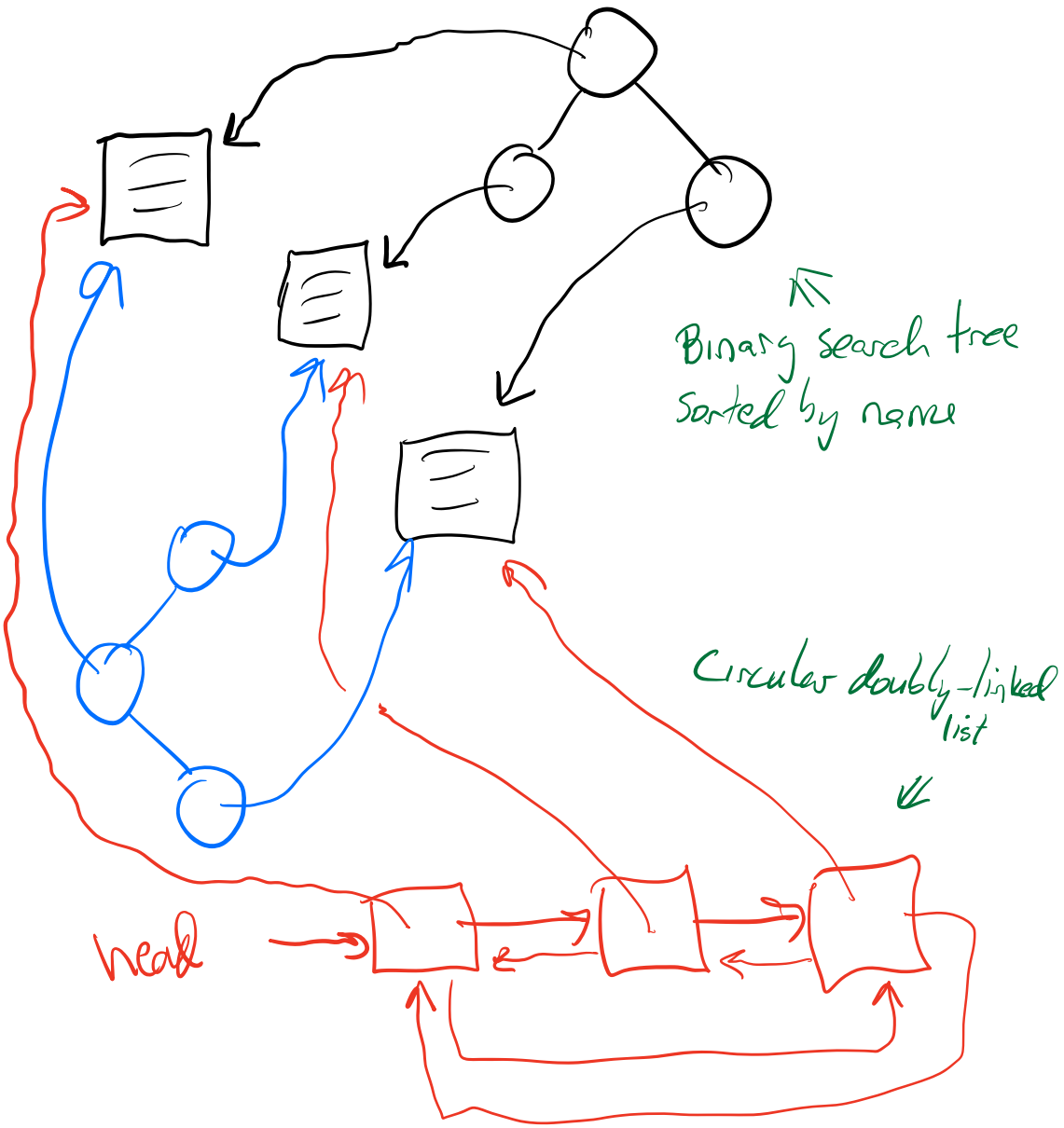


Circular Doubly-Linked List



In the assignment, you'll read in a bunch of personnel records

- name, age, salary, id num.


Binary search tree
sorted by name
Circular doubly-linked list
head

Back to pointers

- In C, you can have pointers to functions!

```
int plus1(int x)
{ return x+1;
}

int times2(int y)
{ return y*2;
}

int main()
{
  int (*p)(int) = plus1;
  printf("%d\n", p(6)); //prints 7
  p= times2;
  printf("%d\n", p(6)); //prints 12
}
```

Parameters can be function pointers:

```
void f(int (*g)(int))
{
  printf("%d\n", g(6));
}
...
f(plus1); // prints 7.
f(times2); // prints 12
```

Using typedef to function pointer type:

```
typedef int (*FTYPE)(int);
```

- defines a type named FTYPE which is a function pointer type.

```
FTYPE p = plus1;
printf("%d\n", p(6)); // prints 7
```

How would I write an increment function:

```
int x=7;
increment(x);
printf("%d\n", x); //prints 8
```

How about this:

```
void increment(int y)
{
  y=y+1;
}
```

No!

C uses "pass by value".
This means that the function call causes a copy of the value of the argument to be passed into the function.

In the above code — within the increment function, y = y + 1;

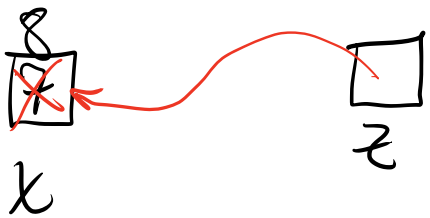
x doesn't change.

So, printing x still prints 7.

The bottom line is that we can't write an increment() function like the one above. Instead, we need to pass the address of x to increment(), and increment() has to dereference that pointer to modify x.

```
int x = 7;
increment(&x);
printf("%d\n", x); //prints 8
```

```
void increment(int *z)
{
  *z = *z + 1; //or (*z)++
}
```



Reading values in from the terminal ("standard input"):

scanf()

Example:

```
int x, y;
scanf("%d %d", &x, &y);
```

must give an address.

```
char name[100];
scanf("%s", name);
```

↑ the name of an array is the address of that array.

```
int a[20];
```

"a" is the same as &(a[0])

Using scanf() to read strings is actually dangerous

- susceptible to a "buffer overflow attack".

- ok for assignment 1, though.

If scanf() finds no more data to read, it returns a constant: EOF

```
while (EOF!=scanf("%d", &x))
    printf("%d\n", x);
```

Built-in string functions

```
#include <string.h>
```

```
strcpy(p1, p2);
```

copies the string pointed to by p2 into the array pointed to by p1
- including the terminating ∅

strlen(p)

- returns the number of characters in the string p.
- not including the terminating ∅.

```
printf("%d\n", strlen("auto"));
    // print 4
```

strcmp(p1, p2)

returns ∅ if p1 and p2 point to strings that are identical.

returns a negative number if p1 points to a string that comes before the string pointed to by p2, alphabetically

otherwise, returns a positive number.

```
int res = strcmp("hello", "there");
if (res == 0)
    printf("same");
else if (res < 0)
    printf("less");
else printf("more");
```

↳ prints "less".