# CS354: Machine Organization and Programming

Lecture 5
Monday the September 14th 2015

Section 2
Instructor: Leo Arulraj

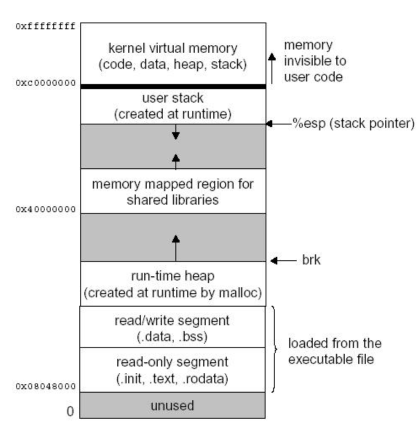


## Class Announcements

1. **Assignment 0's due date was today 9 AM!**
2. Some of you sent email about midterm conflicts and I have noted you down. If you have an exam conflict, email me ASAP.
3. Links to relevant notes from previous versions of this course have now been posted in the lecture schedule. Read them when you get time.
4. Handouts page has a few links to notes on pre-requisite material now. Read them if you don't have prerequisite background.

## Lecture overview

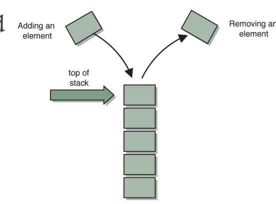
1. Memory layout recap
2. Stacks
3. Linkedlists



# Example C Program on Memory Allocation

## Stack

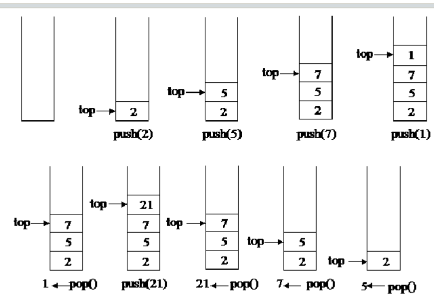
1. LIFO : Last In First Out
2. Data is both added and removed from one end i.e. the top
3. Add/Push: place an item on the top
4. Del/Pop: remove an item from the top




## Common Operations on Stack

1. push(item) : adds item to top of stack
2. pop() : removes item from top of stack
3. top() : peeks into item at stack top
4. is_empty() : whether stack is empty or not
5. size() : number of items in stack

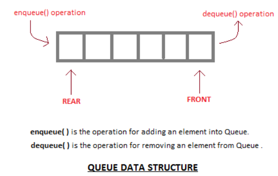
## Stack



## Example C Program on Stack using arrays

## Queue: A mention

1. FIFO : First In First Out
2. Once an element is added to the queue. All elements that were added before it must be removed before the newly added element can be removed.




## Stacks and Functions(Brief Intro)

In today's lecture, we will look at a brief intro about how stacks are used for Function Calls.

We will look at more details in a future lecture.

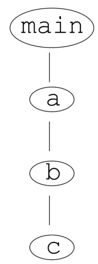
```
int main(    ) {
   x = a();
}

int a() {
   y = b();
}
```

```
int b() {
   z = c();
}

int c() {
}
```

1

main calls a
a calls b
b calls c
c returns to b
b returns to a
a returns to main

The call tree:




2

Partial assembly code for `main()` and `a()`

```
main:     .
          .
      branch  a            # call
a_rtn: copy   x, a_rtn_value
          .
          .
a:        .
          .
      branch  b            # call
b_rtn: copy   y, b_rtn_value
          .
          .
      branch  a_rtn     # return from a
```

3

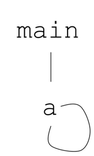
*Modified* assembly code for `main()` and `a()`

```
main:     .

      branch  a            # call
a_rtn1: copy   x, a_rtn_value
          .
      branch  a     # another call to a
a_rtn2: copy   x, a_rtn_value
          .
a:        .
          .
      branch  b            # call
b_rtn: copy   y, b_rtn_value
          .
          .
      branch            # return from a
```

4

## Different example: Recursion

```
int main( ) {
   x = a( );
}

int a(  ) {
   z = a( );
}
```




5

```
main:

        la      a_rtn, rtn1
        branch  a               # call
rtn1:   # copy out return value

a:
        # recursive call
        la      a_rtn, rtn2
        branch  a
rtn2:   # copy out return value
         .
         .
        branch  (a_rtn)
```

6

To make this work, the code needs to

- just before each call, save the return address
- before each return, get and use the most recently saved return address

The data structure required is called a stack.

7

## Different example: Recursion

```
main:

        la      a_rtn, rtn1
        branch  a               # call
rtn1:   # copy out return value

a:      push    a_rtn
        # recursive call
        la      a_rtn, rtn2
        branch  a
rtn2:   # copy out return value
         .
         .
        pop      a_rtn
        branch  (a_rtn)
```

13

The problem of overwritten return addresses is solved with a stack.

The same problem exists with

- parameters
- variables local to a function

The solution bundles all these saved/restored values into a single set to be pushed/popped.

The set of items is called an activation record (AR) or stack frame.

14

Important detail:

A compiler cannot always know if there *is* recursion in a program!

Consider separate compilation.

In file 1:

```
int a() {
    z = b();
}
```

In file 2:

```
int b() {
    x = a();
}
```

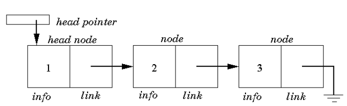
15

# Disadvantages of Arrays

1. Size of an array is fixed.
2. So, programmers allocate array that are large enough to hold the maximum needed in any run. (e.g. stack using arrays program earlier)
3. Shifting elements to make space for new elements at the front of an array is expensive
4. Linked list can solve these issues.

# Singly Linked List

1. Linked list is made up of nodes.
2. Each node points to the next node.
3. The first node is called "**head**" of the linked list.
4. The last node is called **"tail"** of the linked list.




A Linked List

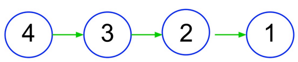
Assume (for simplicity) that the item to go into the list is an `int`.

Start with an empty list.

| | |
|---|---|
| `listadd(1)` | 1 |
| `listadd(2)` | 2 … 1 |
| `listadd(3)` | 3 … 2 … 1 |
| `listadd(4)` | 4 … 3 … 2 … 1 |

"Knowing" where the next item in the list is is simple -- it is a pointer.

We need to associate each item in the list with a pointer.



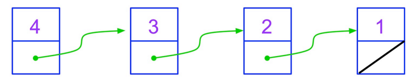

Set up a struct with 2 fields:

int

pointer to a struct

(often called a node)

After adding all 4 ints to the example list:




```
struct node {
    int theint;
    struct node *next;
};
```

**Singly linked, but in the reverse order (add to end or back of the list)**