COMP20003 Workshop Week 5

Tree Traversal
AVL & Rotations

Stacks & Queues

Practical Issues:
- defining a tree node, and a tree or bst
- bst_insert
- stack/queue implementation using linked lists

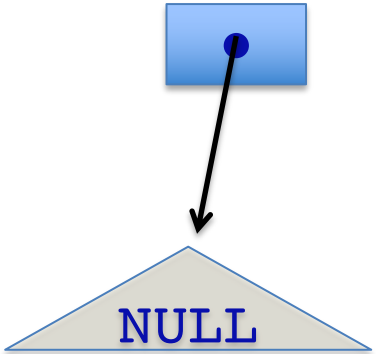
Note: For reference, download this `week5.pdf` from `github.com/anhvir/c203`

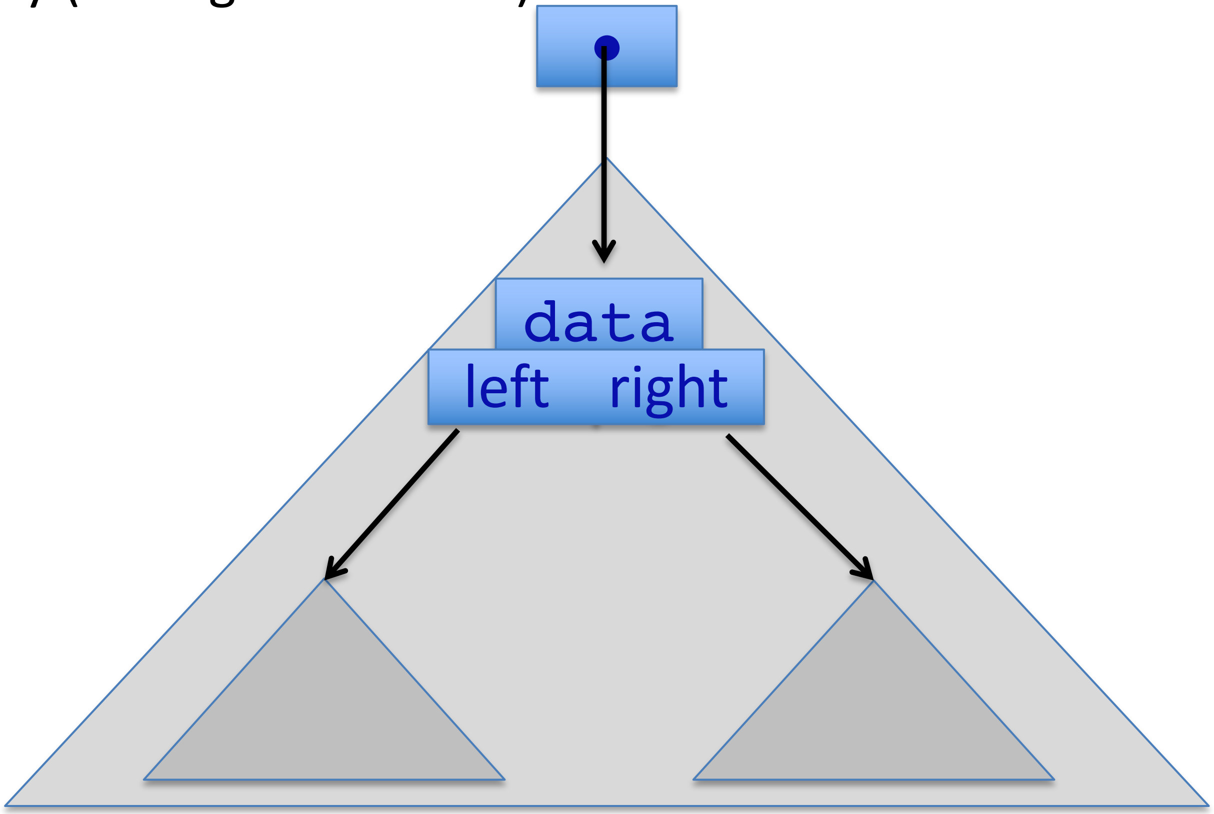
Review: Binary trees and BST

A Binary Tree can have up to 3 components:

- a *root node* (with some `data`) which is connected to:
- a *left sub-tree* (which is a tree), and a
- a *right sub-tree* (which is another tree).

It is convenient to defined a tree as a pointer to its root node. In the base case, a tree is empty (having NULL value).





Declaring trees: code examples

Model 1: in JupyterLab	Model 2:	Notes
`struct bst {`	`typedef struct t_node *tree_t;`	
	`struct t_node {`	a tree node
`struct data *data;`	`data_t *data;`	a data
`struct bst *left;`	`tree_t left;`	left chilld
`struct bst *right;`	`tree_t right;`	right child
`};`	`};`	
`struct bst *t= NULL;`	`tree_t t= NULL;`	at the start, t is empty

Note that often `data_t` include a special field `key`. And:

In programming practice:

- the most convenient way is to declare data as `void *data`

In lecture/workshop demonstrations:

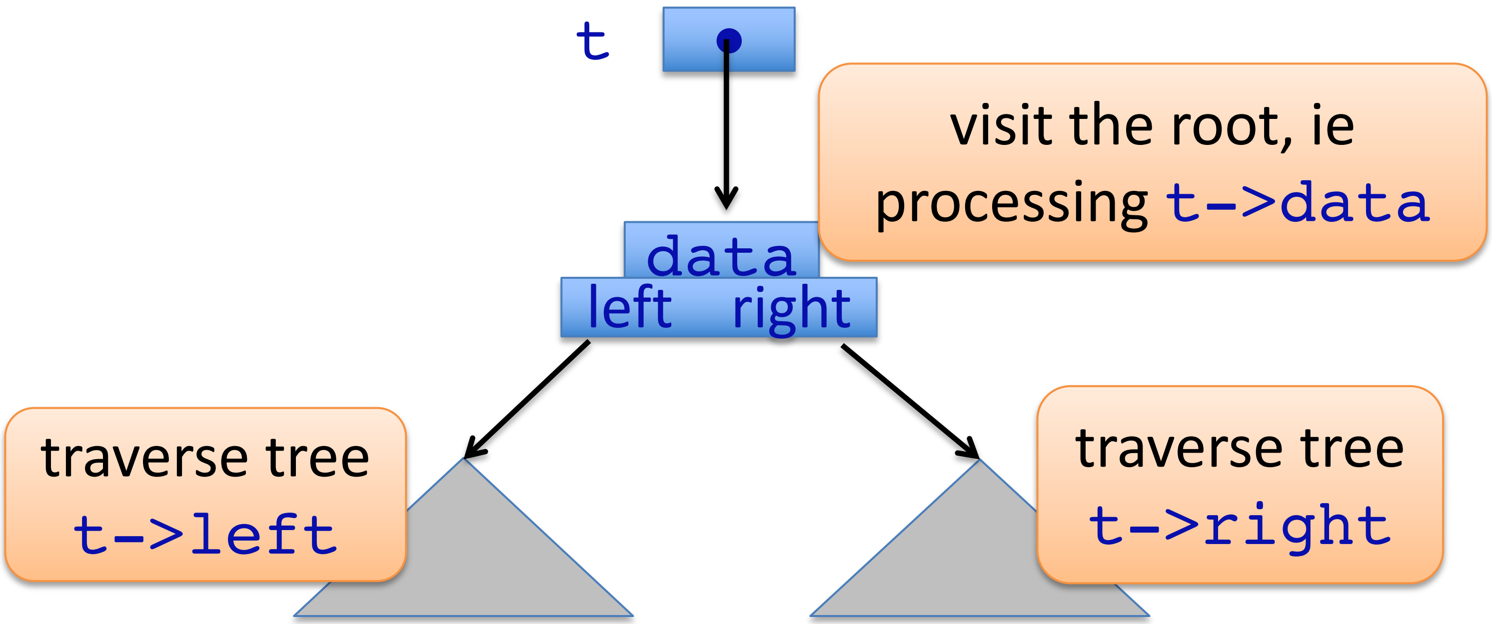
- the most convenient way is to declare data as `int key` (and ignore other components of data)

Tree/BST traversal

Visit a non-empty tree node= performing some actions on the data of the node (such as printing, comparing key with something).

Tree traversal= visit all non-empty nodes of a tree in a systematic way.

For a non-empty tree , there are 3 works , and they can be done in any order!



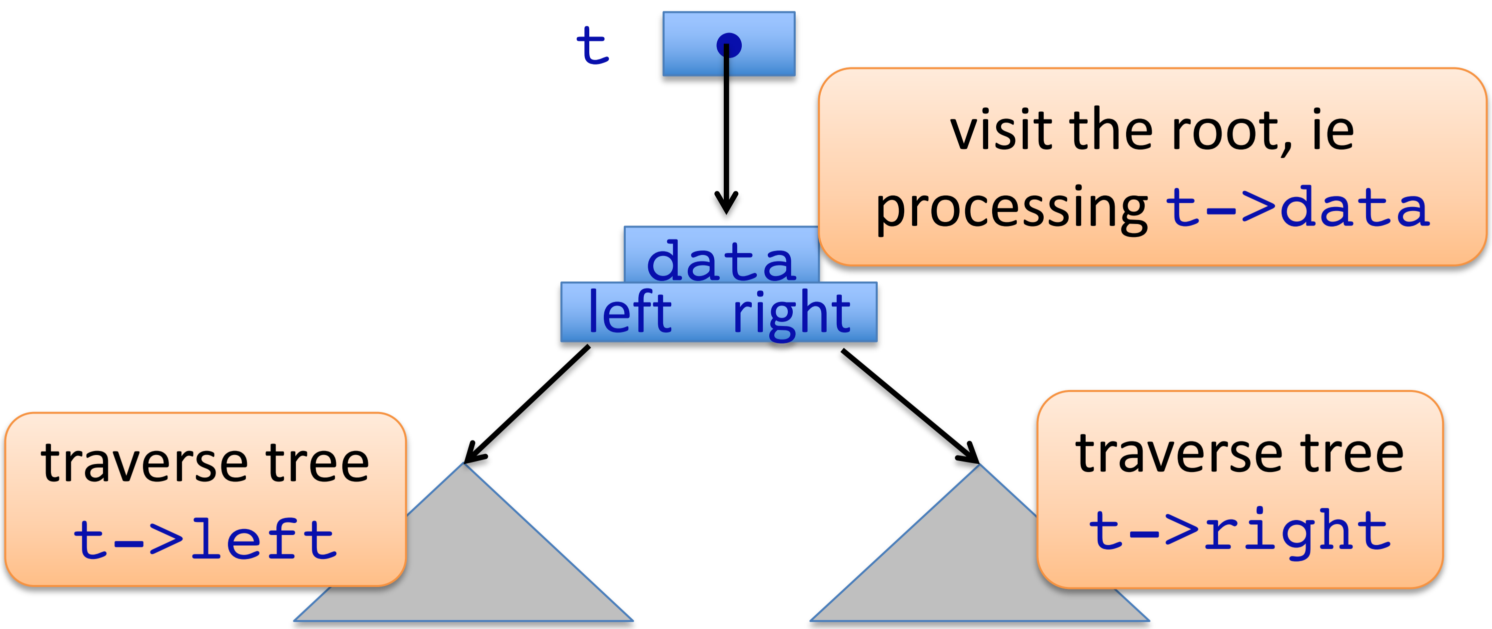
Tree/BST traversal

Depending on when to visit the root node, we have *pre-order* (visit_root first), *post-order* (visit_root last) and *in-order* (visit_root in the middle).

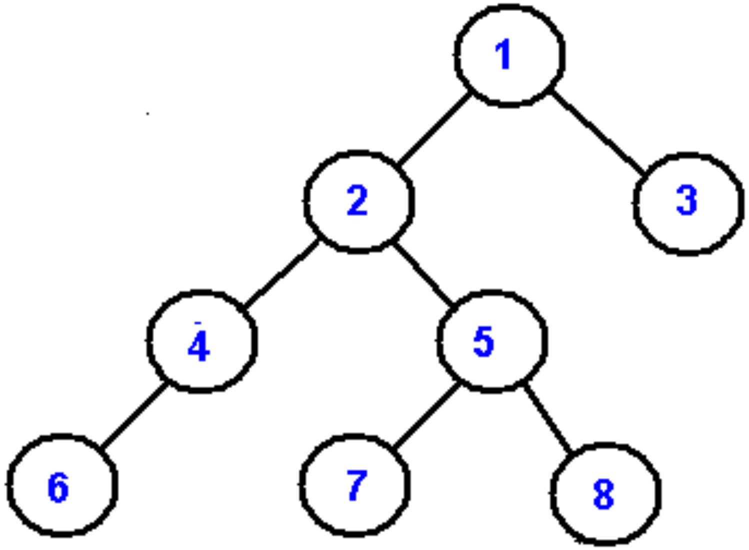
```
void inorder(struct bst *t)
{




}
```



Tree Traversal Examples



List the nodes in order visited by:

- in-order :
- pre-order :
- post-order :

Tree Traversal Exercise (supposing data is just `int key`)

```
struct bst {
   int key;
   struct bst *left;
   struct bst *right;
};
```

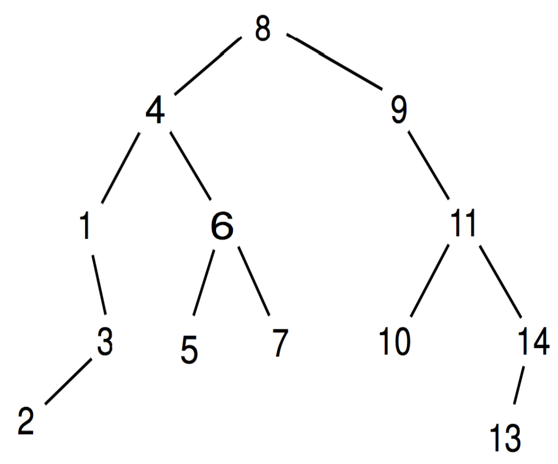
Write a C function for:

- printing a BST's keys in increasing order
- printing a BST's keys in decreasing order

```
    printBST(                    ) {



}
```

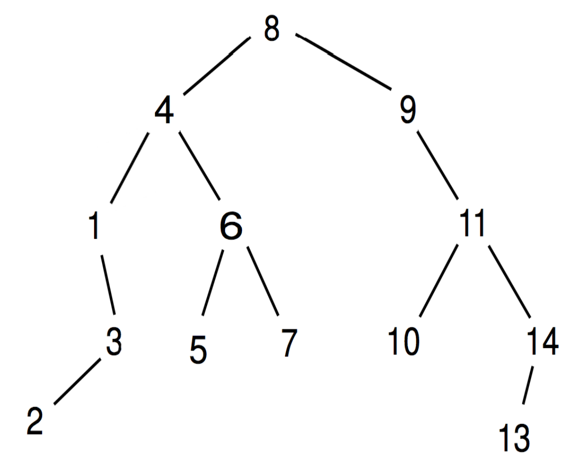


Tree Traversal Exercise (supposing data is just `int key`)

```
struct bst {
   int key;
   struct bst *left;
   struct bst *right;
};
```

What traversal order should be used for:

- copying a tree ?
- free a tree ?

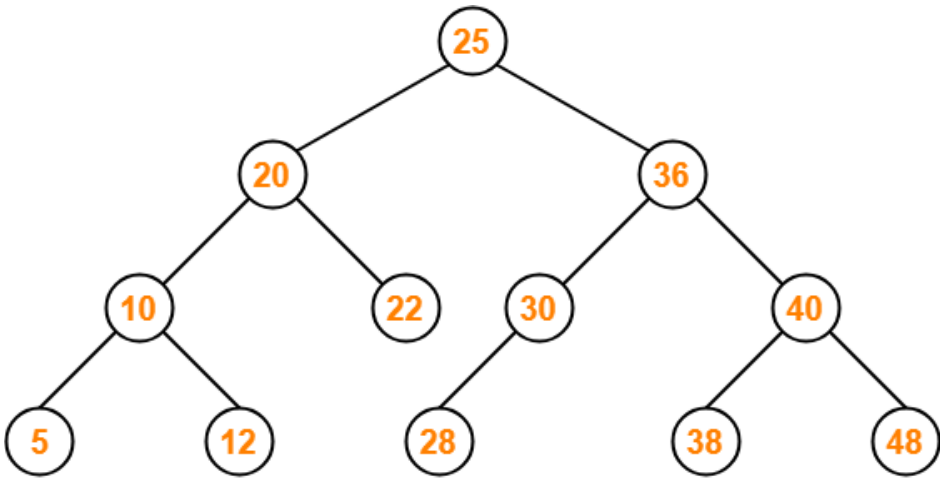


The Goods and the Bads of BST

The Goods:

Average performance for both `insert` and `search` is

`O(log n)`

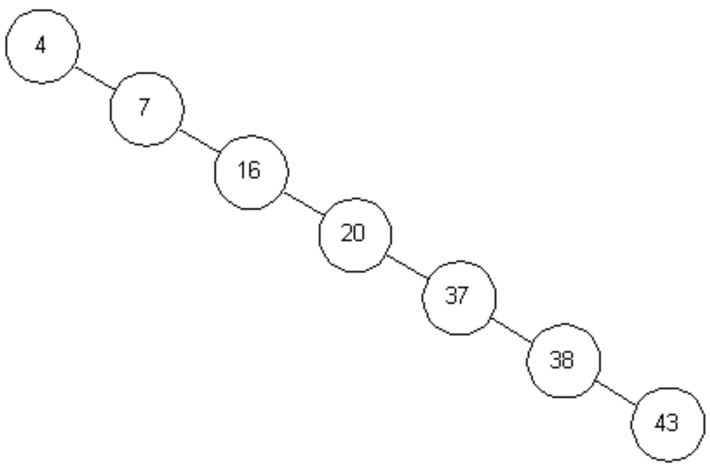


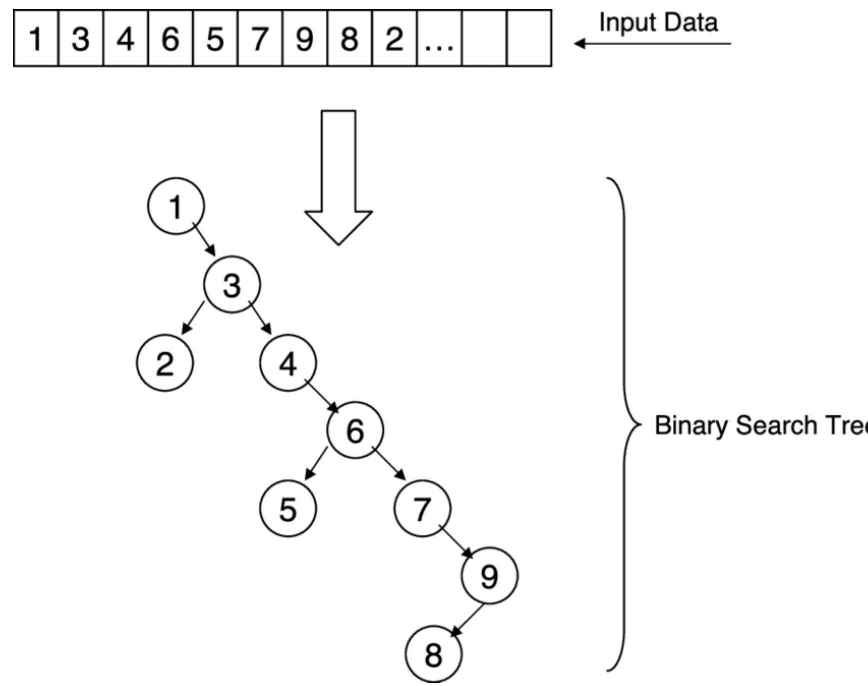
Binary Search Tree

The height of the tree is around $\log_2 n$ in average

The Goods and the Bads of BST

The Bads: *Worst-case* performance for both `insert` and `search` is `O(n)`





The height of the tree could be around `n`

AVL = keeping BST tree balanced

Evolution? Concrete data structures for dict

Operation	Case	Arrays / Linked Lists	Sorted Arrays	BST	AVL
Insert	Average	O( )	O( )	O( )	O( )
	Worst	O( )	O( )	O( )	O( )
Search	Average	O( )	O( )	O( )	O( )
	Worst	O( )	O( )	O( )	O( )

Concrete data structures for dict

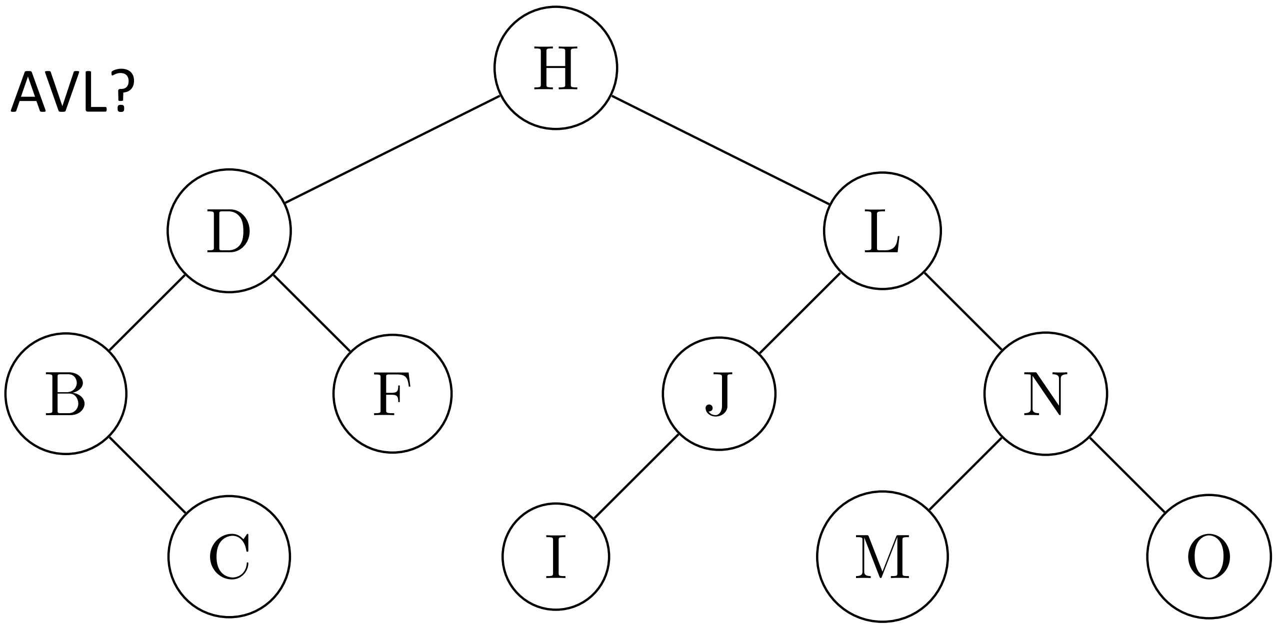
Operation	Case	Arrays / Linked Lists	Sorted Arrays	BST	AVL
Insert	Average	O(1)	O(n)	O( )	O( )
	Worst	O(1)	O(n)	O( )	O( )
Search	Average	O(n)	O(log n)	O( )	O( )
	Worst	O(n)	O(log n)	O( )	O( )

Concrete data structures for dict

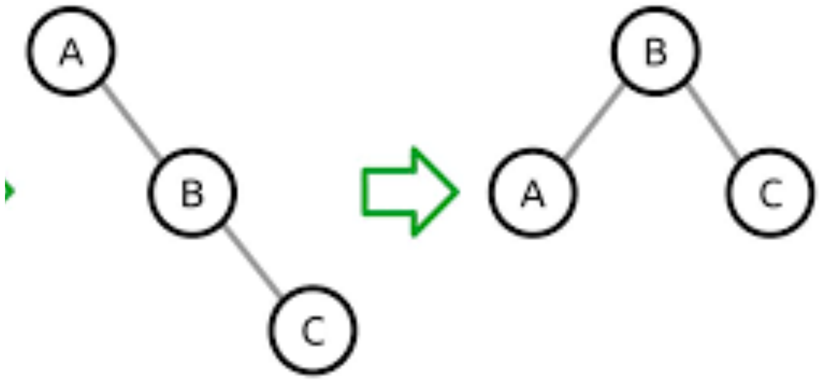
Operation	Case	Arrays / Linked Lists	Sorted Arrays	BST	AVL
Insert	Average	O(1)	O(n)	O(logn)	O(logn)
	Worst	O(1)	O(n)	O(n)	**O(logn)**
Search	Average	O(n)	O(log n)	O(logn)	O(logn)
	Worst	O(n)	O(log n)	O(n)	**O(logn)**
Search overall		O(n)	Θ(logn)	O(n)	Θ(logn)

How to know if a node/tree is imbalanced? Balance factor

Is an AVL?

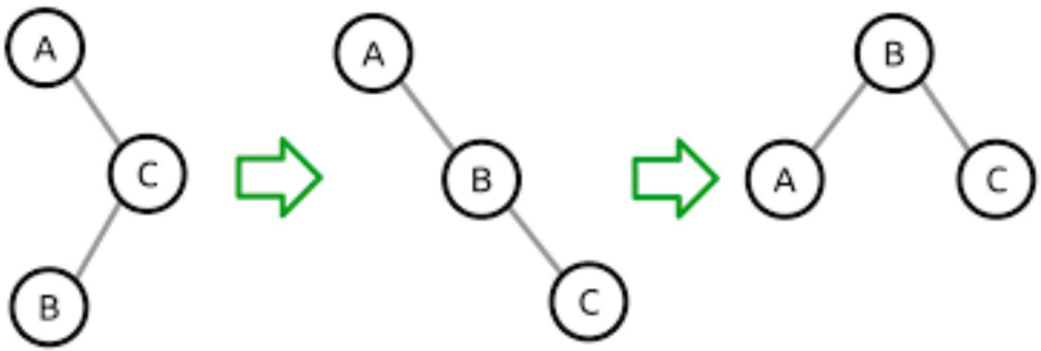


Two Basic Rotations: 1) Single Rotation for Sticks



single **L rotation** applied to root→right_child
single **R rotation** applied to root→right_child

Two Basic Rotations: 2) Double Rotation for non-stick



1) a single rotation to child→grandchild to turn root to a stick
2) another single rotation to balance the stick

Here we have RL double rotation

Using Rotations to rebalance AVL

Problem: When inserting to AVL, it might become unbalanced

Approach: Rotations (Rotate WHAT?, and HOW?)

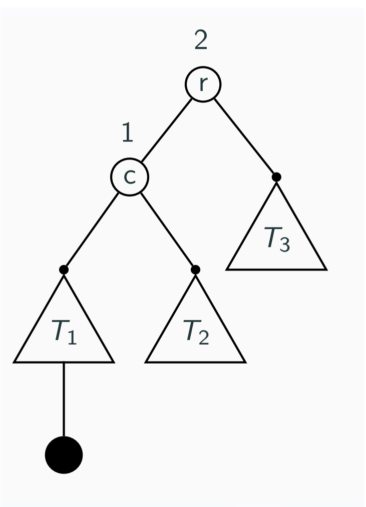
Rotate WHAT?

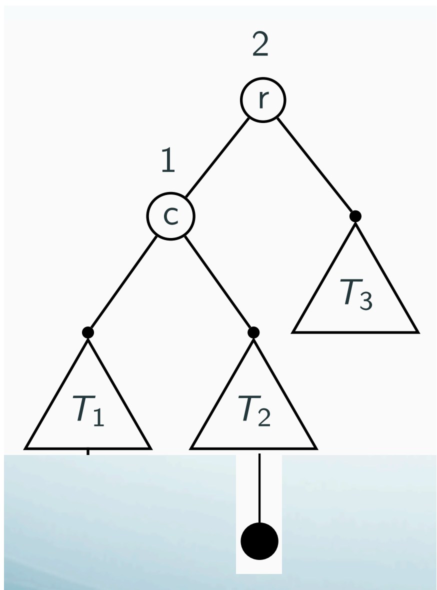
The *lowest* subtree X which is unbalance

HOW

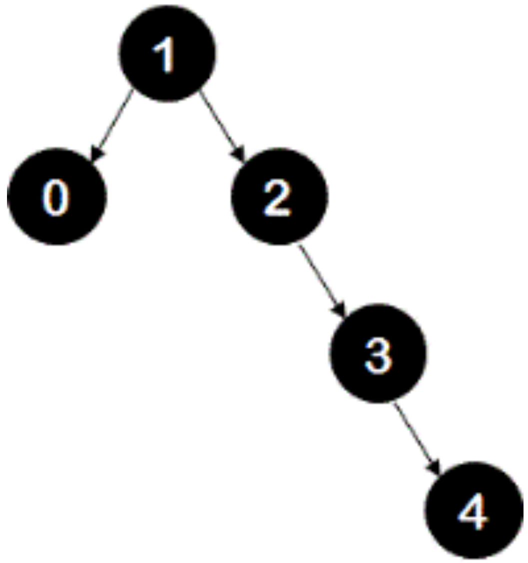
Consider *the first 3 nodes* X→A→B in the path from root X to the just-inserted node

Apply a single rotations if that path is a stick, double rotation otherwise

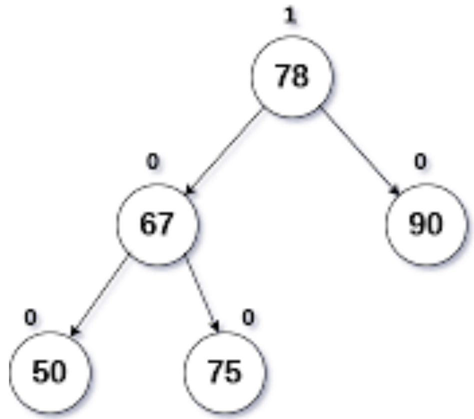


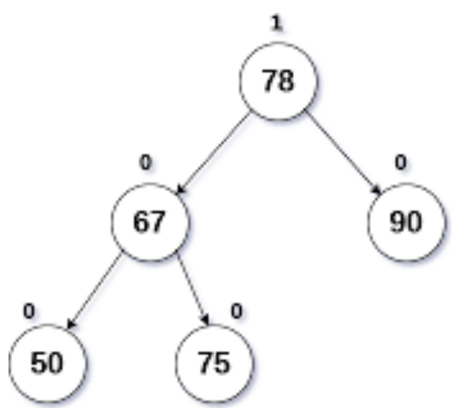


Examples

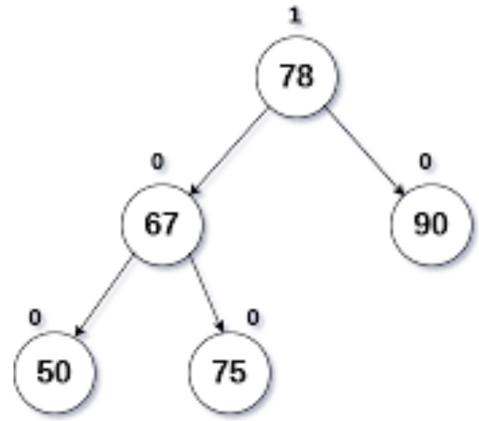


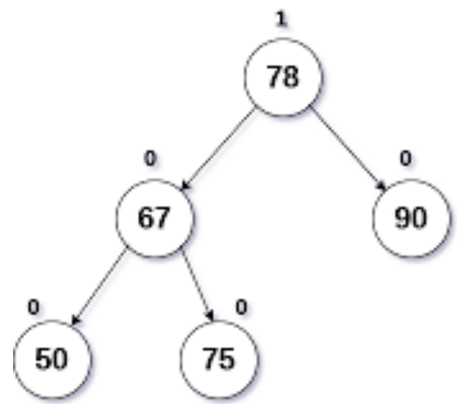
Examples: rebalance after inserting 60?   70 ?





Examples: rebalance after inserting 60?   70 ?





Programming : How to implement bst_insert?

Input: tree `t` (`struct bst *t`; OR `tree_t t;`)

`key`

Output: a new node with `key` is inserted to `t` in a right position

Function header for `struct bst *t`

```
???    bst_insert( ???     );
```

How to implement bst_insert?

Is this function correct ? Supposing `t` is `NULL` at the beginning.

```
struct bst *bst_insert(struct bst *t, int key) {
  if (t==NULL) {
    t= malloc(*t);
    t->key= key;
    t->left= t->right= NULL;

  } else if (key < t->key)
              bst_insert(t->left, key);
  else …
}
```

Notes: For `tree_t`, replace `struct bst*` with `tree_t:`

How to implement bst_insert? Version 1

Is this function correct ? Supposing `t` is `NULL` at the beginning.

```
struct bst *bst_insert(struct bst *t, int key) {
  if (t==NULL) {
    t= malloc(*t);
    t->key= key;
    t->left= t->right= NULL;
    return t;
  } else if (key < t->key)
            return bst_insert(t->left, key);
  else …
}
struct bst *t= NULL;
t= insert(t, 10);
```

Notes: For `tree_t`, replace `struct bst*` with `tree_t:`

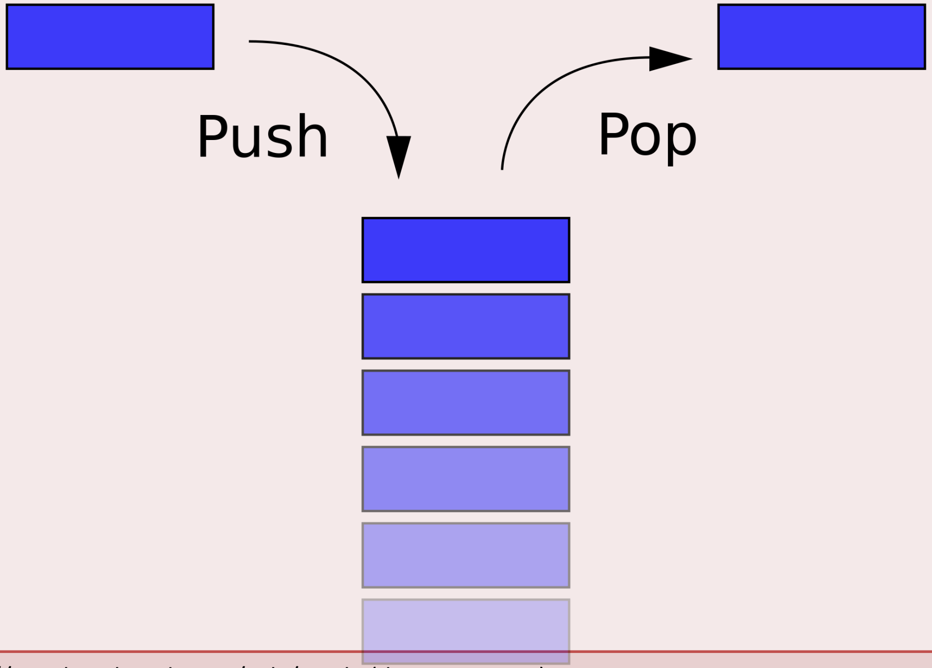
How to implement bst_insert? vesrion 2

Is this function correct ? Supposing `t` is `NULL` at the beginning.

```
void bst_insert(struct bst **t, int key) {
  if (*t==NULL) {
    *t= malloc(**t);
    (*t)->key= key;
    (*t)->left= (*t)->right= NULL;
  } else if (key < (*t)->key)
            bst_insert(&((*t)->left), key);
  else …
}
```

Notes: For `tree_t`, replace `struct bst*` with `tree_t:`

ADT: Stack (LIFO)



http://www.123rf.com/stock-photo/tyre.html

https://simple.wikipedia.org/wiki/Stack_(data_structure)

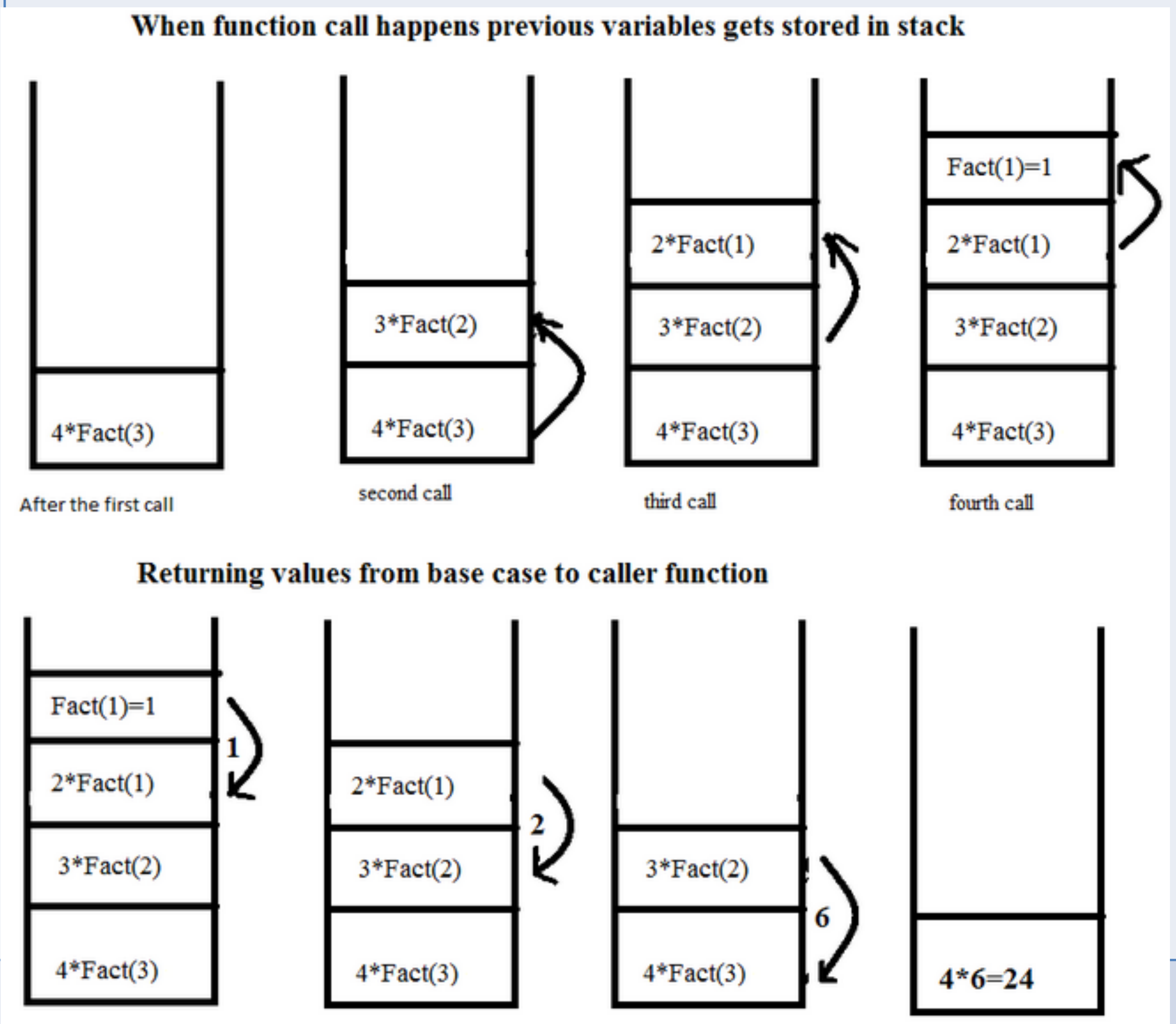
Stack Operations	`push()` : add an element into stack
	`pop()` : remove an element from stack
	`isEmpty()` : check if stack is empty
	`newList()`
	`freeList()`

Stack is widely used in implementation of programming systems. For example, compilers employ stacks for keeping track of function calls and execution.

Stack for :

fact(4)

```
int fact( int n ) {
 if ( n<=1 )
     return 1;
 return n*fact(n-1);
```



Other applications?

Q5.1: Stack Implementation

Using arrays? How? What should be considered?

Using linked lists? How? What should be considered?

Make sure that you can use arrays or linked lists to implement stacks so that:

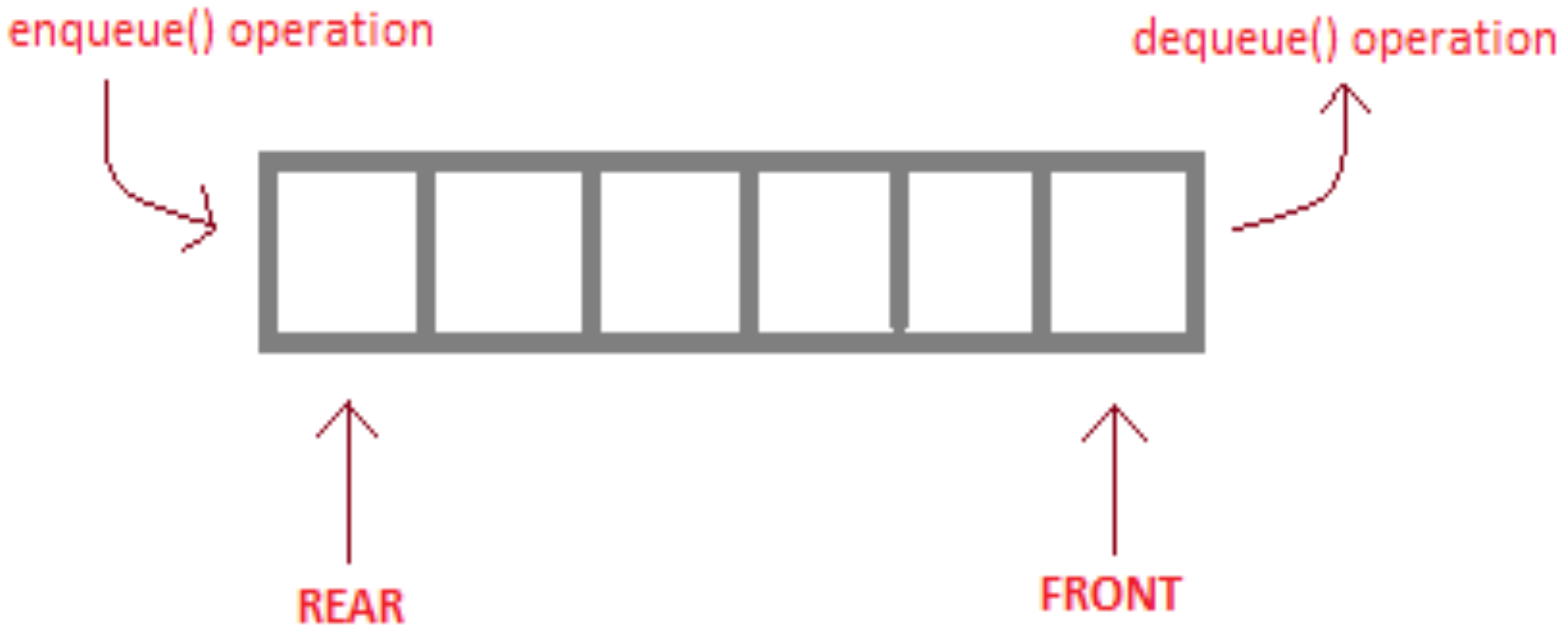
- both `push` and `pop` have complexity `O(1)` (or amortised `O(1)`)

Stacks: Array Implementation

```
#define INIT_SIZE 8
typedef struct {
    int *a;    /* supposing that stack's elements are int */
    int head;
    int size;
} stack_adt;
void createStack( stack_adt *ps) {
    ps= malloc(sizeof(*ps));   // and assert(…
    ps->size= INIT_SIZE;
    ps->a= malloc( ps->size * sizeof(int) ); // assert
    ps->head= 0;
}
void push( stack_adt *ps, int x) {
    if ( ps->head == ps->size ) {
        ps->size *= 2;
        ps->a= realloc( ps->size * sizeof(int) ); //assert
    }
    ps->a[ ps->head++ ]= x;
}
int pop( stack_adt *ps ) {
    assert( ps->head > 0 );
    return ps->a[ --ps->head ];
}
void deleteStack( stack_t *ps )  {   ???  }
```

Stacks: Linked List Implementation

ADT: Queue (FIFO)



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

Data structure: Queue (FIFO)

Queue Implementation

Using arrays?

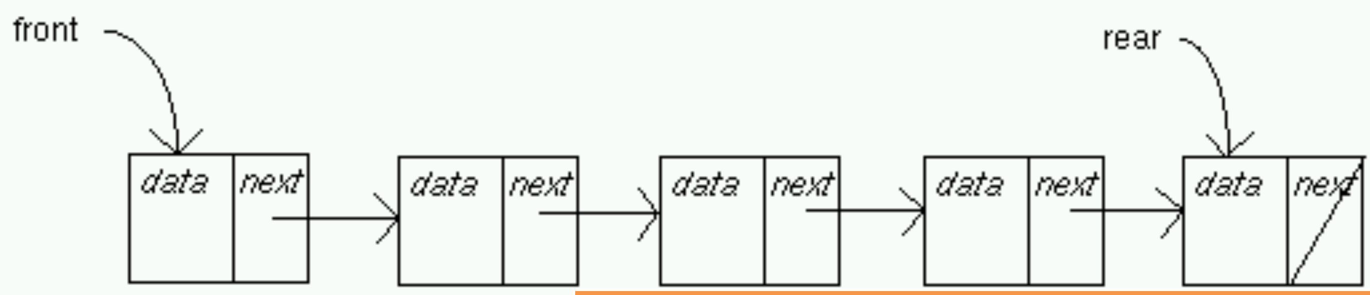
Using linked list?

Any problem with arrays? with lists? Which one is easier?

Make sure that you can use arrays or linked lists to implement queues so that:

- both `enqueue` and `dequeue` have complexity `O(1)` (or amortised `O(1)`)

Queues & Linked Lists



llist.h

```
typedef struct {
   data_t *d;
    struct node *start;
    struct node *end;
} list_t;
list_t *newList();
list_t *insert_at_start
    (list_t *l, data_t *d);
list_t *insert_at_end
    (list_t *l, data_t *d);
//  you can employ Alistair's list.c if you like
```

queue.h

```
typedef list_t q_t;
q_t *newQueue();
q_t *enqueue(q_t *q, data_t *d);
```

queue.c

```
q_t *newQueue(){
  return newList();
}

q_t *enqueue(q_t *q, data_t *d) {
  return insert_at_end(q, d);
}
```

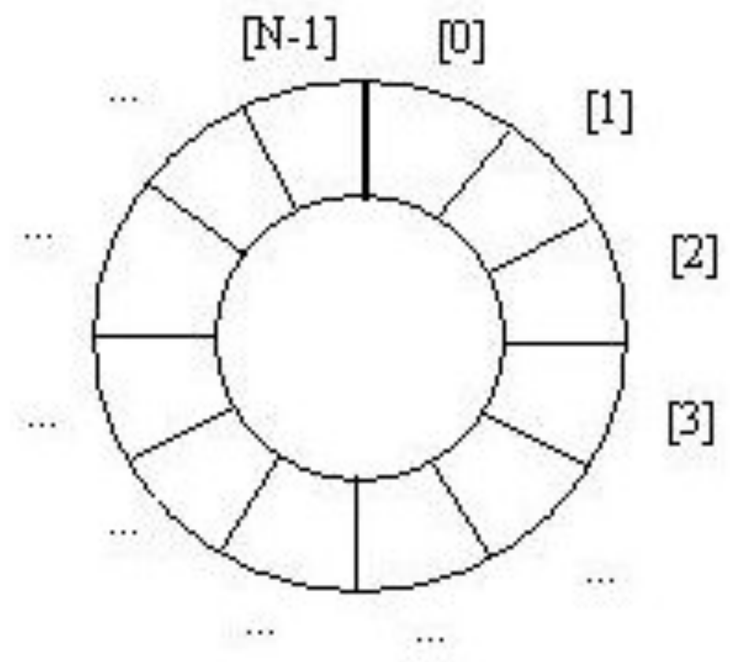
array implementation?

Make sure you can have O(1) for both `enqueue` and `dequeue` (similar to linked list implementation)

Queue Implementation

Using circular arrays?

Circular arrays: 1D arrays, where the first element is considered as next to the last element. Hence, the element next to a[i] is a[(i+1) mod N]



Queue Implementation

Potential problems with circular arrays:

- static circular arrays?
- dynamic circular arrays?

Lab: JH Week 4. For JH Week 5: use Terminal and files in workshops/week5

- BST's insert and free: Implement function `bstInsert` and `freeTree` under *Question 3.1* of *JH Week 4*. Run and make sure that they are correct. Notes: This task is more important than the exercises in Week 5.
- *Tutorial Questions 4.1* and *4.2* in *JH Week 5*
- *Programming Exercises 4.1* in *JH Week 5* (implementing stacks using linked lists), using `Terminal` and files in `workshops/week5/`

`Note 1: It's better just to use your linked list implementation, and build module stack based on module list.`

`Note 2: You also can use Alistair's linked list:`