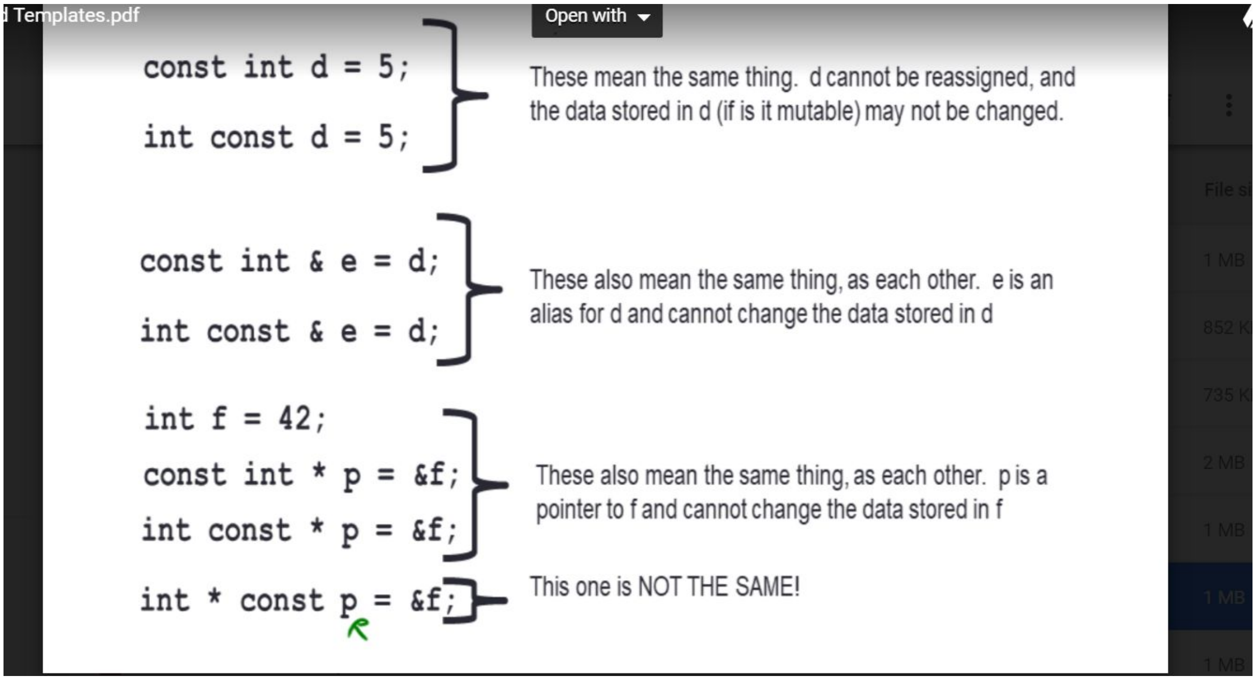
CSE100 Midterm 1 Review

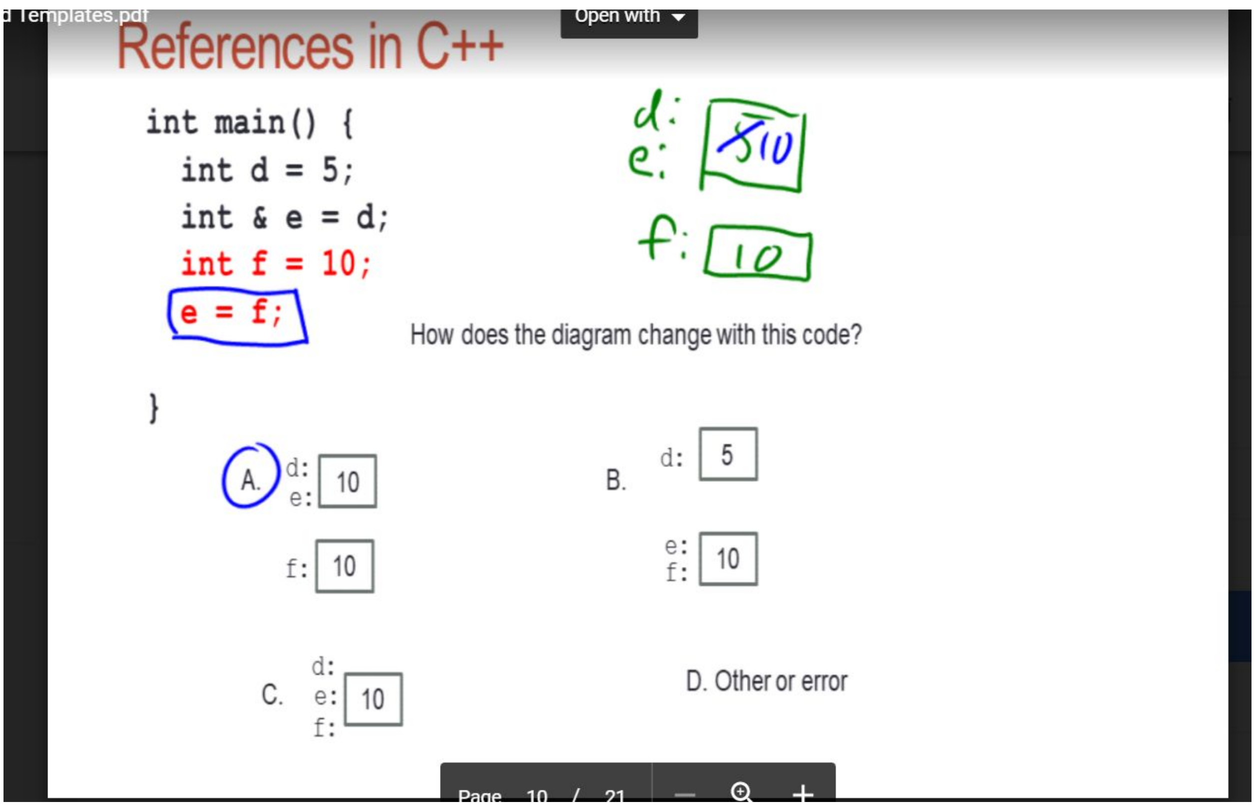
This review doc aims to provide a concise summary of the concepts, algorithms and lecture slides covered in CSE110. *Created by M.*, feel free to collaborate.

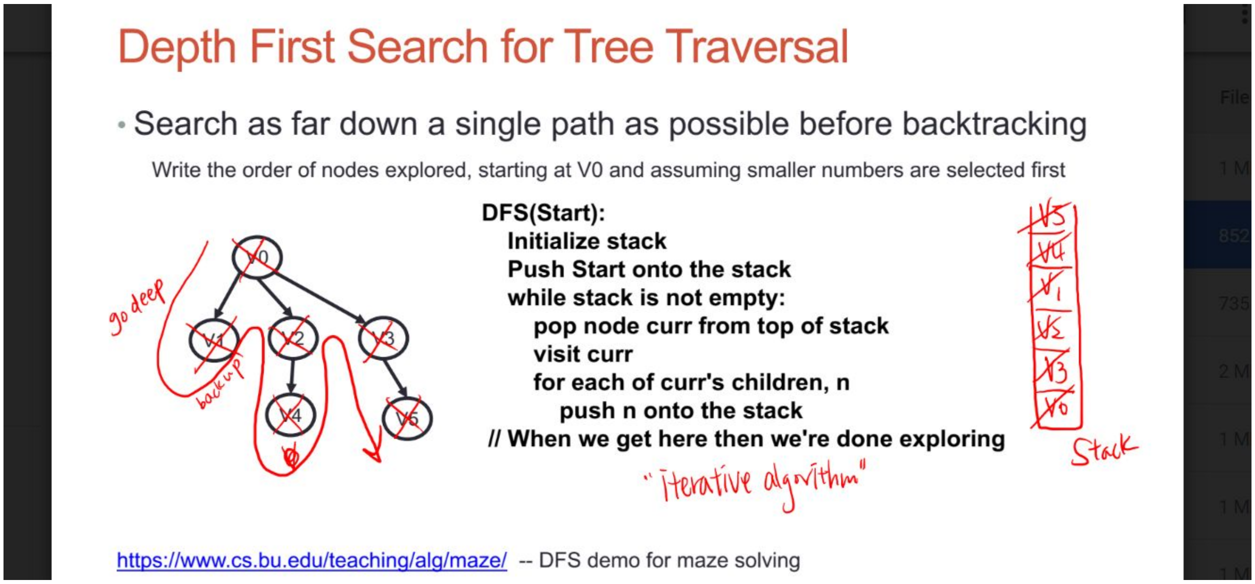
<u>Topics</u>

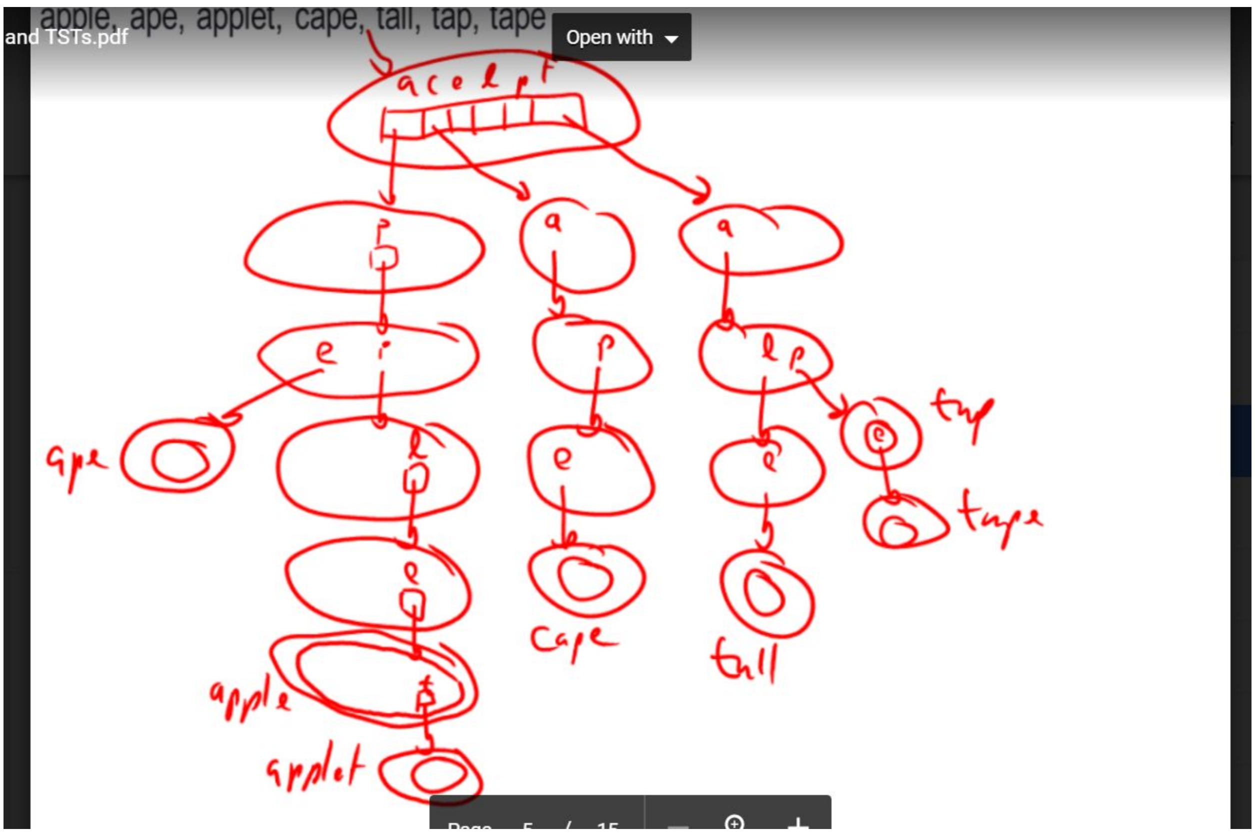
- Parameter passing and assignment is done by **value, by default.**
    - Use reference if want to avoid copy
- Creating an object on stack could be dangerous
    - Destroyed once exit the method
    - If on stack, do **NOT** try to pass to, or return (most case)
- Declaring Node left, right, and parent without using **pointers** could cause compile error
    - Compiler does **NOT** know how much memory to allocate
    -
- **BST**
    - **One node tree has height 1 in this class**
    -
- **KDT**
    - left.dim < root.dim ≤ right.dim
    - Select the **leftmost** node as mid point
    - findNN
        -
- Benchmarking
- Total depth of a tree is **the sum of the depth of each node**
- **TST**
    - If matches a letter, search down the middle child
    - Slower than **MWT**, but more space-efficient
    - Less space-efficient than **BST**, but faster
- **Hash**
    - Property of **equality** (must be hold)
    - Property of **inequality** (not necessary tho)
    - **the average-case performance of a Hash Table is *independent* of the number of elements it stores -** O(1)
    - Capacity to be prime number
        - Avoid of unequal distribution of elements
    - Design
        - Specifically, if we expect to be inserting *N* keys into our **Hash Table**, we should allocate an array roughly of size ***M* = 1.3*N***
        - Make sure the **load factor** never exceeds 0.75
    - Deal with collision
        - Separate chaining
            - Could affect the runtime of insert and find
        - Linear probing

- Could cause the failure of removing?
- Double hashing (not covered yet)



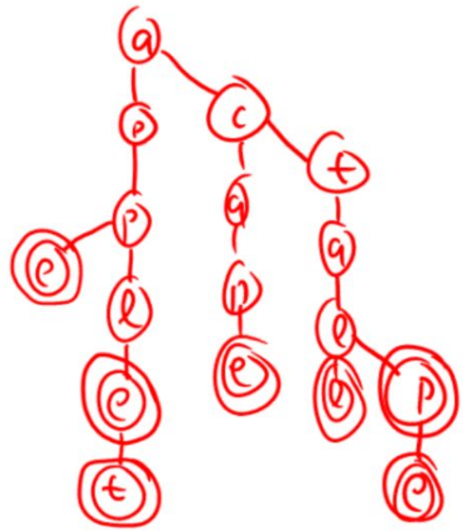



Depth First Search for Tree Traversal
• Search as far down a single path as possible before backtracking
Write the order of nodes explored, starting at V0 and assuming smaller numbers are selected first
V0
V1
V2
V3
V4
V6
go deep
back up
DFS(Start):
Initialize stack
Push Start onto the stack
while stack is not empty:
pop node curr from top of stack
visit curr
for each of curr's children, n
push n onto the stack
// When we get here then we're done exploring
"iterative algorithm"
V3
V4
V1
V2
V3
V0
Stack
https://www.cs.bu.edu/teaching/alg/maze/ -- DFS demo for maze solving


apple, ape, applet, cape, tall, tap, tape
and TSTs.pdf
Open with
a c e l p t
ape
apple
applet
cape
tall
tap
tape

Draw the ternary search trie for the following (in this order)

apple, ape, applet, cape, tall, tap, tape



A Big-O Challenge

```
void tricky(int n) {
    int operations = 0;
    while(n > 0) {
        for(int i = 0; i < n; i++) {
            cout << "Operations: " << operations++ << endl;
        }
        n /= 2;
    }
}
```

$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8}$

$2n$ $\quad O(n)$

What is the *tightest Big-O bound* for the code above?

A. O(log n)
B. O(n)
C. O(n * log n)

If $\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = c > 0$ and finite, then $\quad f(n) \in \Theta(g(n))$

If $\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = c$ is *finite*, then $\quad f(n) \in O(g(n))$

If $\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = c > 0$ or *infinite*, then $\quad f(n) \in \Omega(g(n))$

Resolving Collisions: Separate Chaining

- using the hash function H(K) = K mod M, insert these integer keys:

701 (1), 145 (5), 218 (1), 12 (5), 750 (1)

in this table:

index:	0	1	2	3	4	5	6
		701 → 218				145 → 12	

Linear probing: inserting a key (Worksheet, problem 2 again)

- When inserting a key K in a table of size M, with hash function H(K)

1. Set indx = H(K)
2. If table location indx already contains the key, no need to insert it. Done!
3. Else if table location indx is empty, insert key there. Done!
4. Else collision. Set indx = (indx + 1) mod M.
5. If indx == H(K), table is full! (Throw an exception, or enlarge table.) Else go to 2.

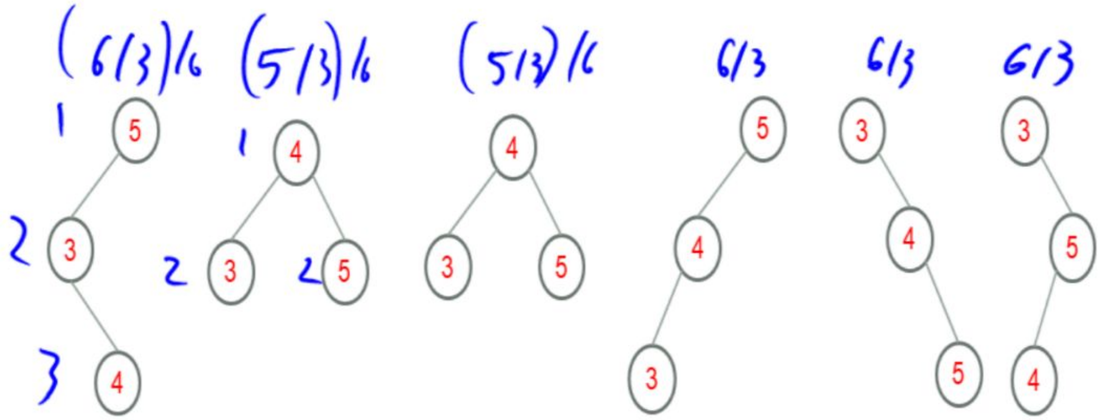
```
int a = 5;                    // create a regular int
int b = 6;                    // create a regular int
const int * ptr1 = &a;        // can change what ptr1 points to, but can't modify the actual data pointed to
int const * ptr2 = &a;        // equivalent to ptr1
int * const ptr3 = &a;        // can modify the data pointed to, but can't change what ptr3 points to
const int * const ptr4 = &a; // can't change what ptr2 points to AND can't modify the actual object itself

ptr1 = &b;                    // valid, because I CAN change what ptr1 points to
*ptr1 = 7;                     // NOT valid, because I CAN'T modify the data pointed to

*ptr3 = 7;                    // valid, because I CAN modify the data pointed to
ptr3 = &b;                    // NOT valid, because I CAN'T change what ptr3 points to

ptr4 = &b;                    // NOT valid, because I CAN'T change what ptr4 points to
*ptr4 = 7;                    // NOT valid, because I can't modify the the data pointed to
```

With **references**, the `const` keyword isn't too complicated. Basically, it prevents modifying the data being referenced via the `const` reference. Below are some examples with explanations:

```
int a = 5;              // create a regular int
const int b = 6;        // create a const int

const int & ref1 = a; // creates a const reference to 'a' (can't modify the value of a using ref1)
                        // This is OK.  Even though a is allowed to change, it's OK to have a reference that
                        // does not allow you to change it because nothing unexpected will happen.
int const & ref2 = a; // equivalent to ref1

ref1 = 7;               // NOT valid, because ref1 can't modify the data

const int & ref3 = b; // valid, because you can have const reference to const data
int & ref4 = b;         // NOT valid, because you CAN'T have a non-const reference to const data
                        // ref4 might change the data but b says it shouldn't be changed, which is unexpected

int & const ref5 = a; // invalid syntax (const must come before the & symbol)
```

What is the average number of comparisons needed to find an element in *any* BST with 3 nodes?

A. 34/6 B. 29/5 C. 3 D. 2 E. 34/18



Find the average # of comparisons needed to find an arbitrary element in a *specific* BST

then

Average this value over all possible BSTs with N nodes

Probabilistic assumption #1: All keys equally likely to be searched for

Probabilistic assumption #2: All insertion orders are equally likely