COMP 120 Review

Basic Data Types

Char
int
float
double

Width and Precision

Specifier	output
%f	24.527810 (9 spots)
%12f	---24.527810 (12 spots)
%12.3f	-------24.528 (12 spots)
%-12.3f	24.528------ (12 spots)

operator	meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Loops

```
while(i<SIZE)
{
 // Statements
}

Do
{
 /*statements*/
}while(i<SIZE);

for(i=0; i<SIZE; i++)
{
 /* Statements */
}
```

Constants

- Variables that are fixed

```
const float pi = 3.14;
```

Type Casting

```
float Salary = 100.26
return (float) salary;
```

I/O Functions

```
Scanf("%4d", &num);
```

- Scans the first 4 digits and assigns to num

```
printf("%9.2f", x);
```

- .2 is digits after decimal
- 9 is width, if greater than x then right justified. If smaller x is printed

Boolean Expressions

*	Multiply
/	Divide
+	Addition
-	subtraction
%%	Modulus // Gives remainder, both must be integers

Operator	meaning
&&	and
||	or
!	not

Axioms

0 && X = 0

1 || X = 1

!0 = 1

If/Else

```
if(a%2==0)
 printf("\n %d is even", a);
else
 printf("\n %d is odd", a);
```

Switch

```
char ch;
switch(ch)
{
 case 'a':
   printf("\n %c is a vowel", ch);
   break;
 case 'b':
   printf("\n %c isn't a vowel", ch);
   break;
}
```

Break

-Breaks out of the most inner loop its nested in

Functions

prototype

```
function_name(Variable 1, Variable 2);
```

for an array:

```
int sum(int arr[], int size);
```

Pointers

```
int a=5      // a has value 5
int *p = &a; // p has value address of a
```

Dereferencing:

p = address

*p = value

Generic pointer:

```
Void *gp;
```

- must be cast ie:

```
printf("\n %d", (int)gp);
```

Pointers

-contains the adress of the object

ex: int a=5;

in int* p=&a;

p = mailbox number

*p = 5

Dynamic Memory

malloc( )

- Allocates memory and returns a pointer to the first spot in memory

Calloc( )

-Allocates space and intilizes to zero

free( )

-frees the memory

-watch for dangling pointers

Linked structures

Strings

-a character array terminated by '\0'

int* i;

i[0] = 3;

*(i+0)=3;

-The name of the string is a pointer to the address of the first character

String length

length = Strlen(mystring);

String compare

if(strncmp(string1, string 2, length)==0

printf("first length letters are the same")

else printf("not the same")

String Searching

result = strstr(string 2, string 1);

if result = Null //not found

String Concatenation

Strcat(a_long_string, "strings");

Structures

typedef struct {

int empnum;

Char name[MAXLEN];

double salary;

} Employee;

accessing

variable_name.member

malloc

int* i;

i = (int*) malloc (10*sizeof(int

free(i);

CPSC 259 Midterm

Topics

- Arrays
- Pointers
- Dynamic Memory
- Strings
- Structures
- Asymptopic Analysis
- Code Analysis

String functions in <string.h>

Strcpy(S1, S2)
- Changes S1 to S2
- copies S2 into S1 replacing S1

Strcat(S1, S2)
- copies S2 S2 onto the end of S1

Strlen(S1)
- retuhs length of S1

strcmp(S1, S2);
- returns
  - -1: $S1 < S_2$
  - 0: $S1 = S_2$
  - 1: $S1 > S2$

Strchr(S1, ch)
- returns a pointer to the first occurence of ch in S1

Strstr(S1, S2)
- returns a pointer to the first occurence of of S2 in S1

Pointers

```
int *pntr=NULL;
//The value of pntr
is zero//
```

How to free dynamic memory

```
int* i = (int*) malloc(sizeof(int));
*i = 5;
free(i);
i = NULL;
```
← if not, dangling pointer

File I/O

```
FILE* filepointer = NULL;
fopen_s(&filepointer, "data.txt", "r");
while(fgets(buffer-string, SIZE, filepointer);
{
   Sscanf(buffer-string, "%s %s", S1, S2);
}
```

- fgets goes to EOF, you would have to printf in the loop but sscanf stores the two strings gapped by a space in S1, S2

Dereferencing

```
*(pntr + 20) = 7;
pntr[20] = 7;
```

Structures

- above int main()

```
Struct Books {
   Char title[50];
   Char author[50];
   Char subject[100];
   int book_id;
};
//Define type Books variables
 Struct Books Book1;
 Struct Books Book2;

//access by .

Strcpy(Book1.title, "C programming");
 Book1.book_id = 25678;
```

*if you pass as a function you use the function parameter name. `.` to open

Typedef

- Same as struct but allows you to put in Book1, Book2 without saying Struct

```
typedef Struct {
     Char title[50];
     Char author[50];
     Char subject[100];
     int book_id;
} Book;

//instead of struct write Book
 Book Book1;
 Book Book2;
```

Pointers to Structures

```
Book* struct-pointer;
 struct-pointer = &Book1
 // to access inside Book1
  struct-pointer->title;
```

Complexity / Analysis

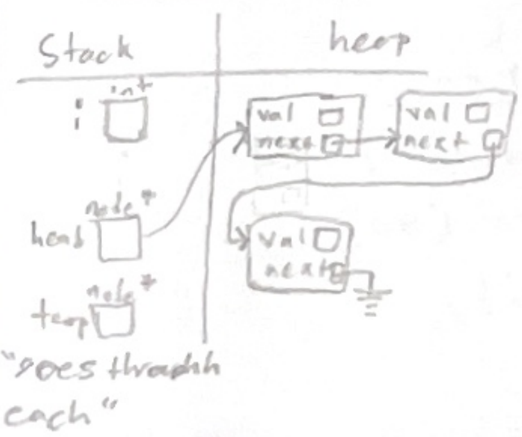
Order of Growth:

Constant	(1)
Logarithmic	O(log n)
Poly-Log	$(\log n)^k$
Linear	n
log-Linear	n log n
quadratic	n^2
Cubic	n^3
Polynomial	n^k
Exponential	c^n

Linked Lists

```
typedef struct node{
    int val;
    struct node* next;
} node;
```

Create a list:




```
for(i=0;i<10;i++)
 {
  temp->next=(node*)malloc(sizeof(node));
  temp->value=i;
 temp=temp->next;
 }
temp->next=NULL;
```

Big O

$0 \le f(n) \le c \cdot g(n)$ for $n \ge n_0$
for some constants c and n_0

Big Ω

$0 \le c \cdot g(n) \le f(n)$ for $n \ge n_0$

Big theta

$c \cdot g(n) \le f(n) \le d \cdot g(n)$ for $n \ge n_0$
different constants

Code Analysis

Linear loops — runs n times $f(n) = n$

```
for(i=0; i<1000; i++)
```

Logarithmic loop — runs $\log_5 n$ times $f(n) = \log_5 n$

```
for(i=1; i<1000; i*=5)
```

Nested loops

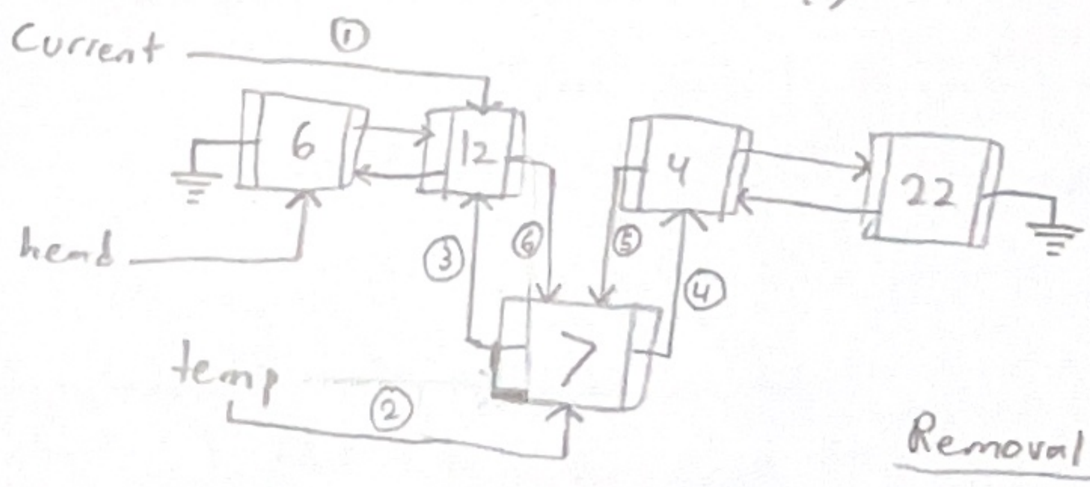
Multiply the loops together

Doubly linked-Lists

```
Struct Node{
  int data
  Struct Node* prev;
  Struct Node* next;
};
```

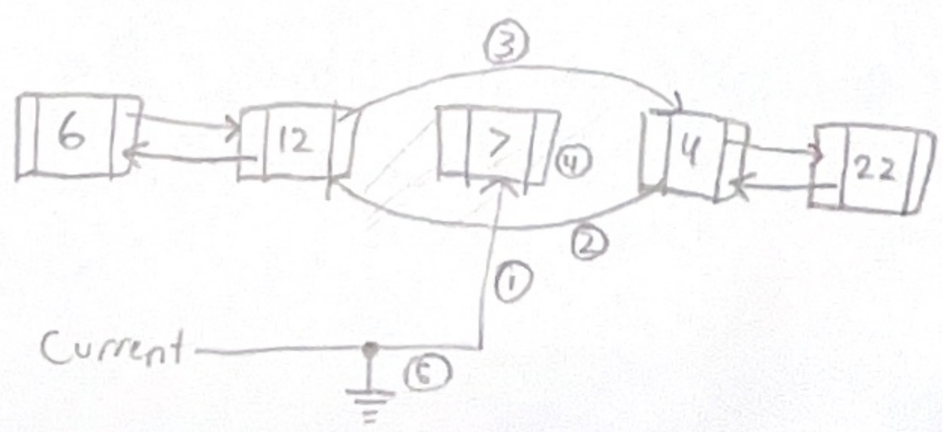
Insertion Algorithm:

```
Node* current, *temp;
① //iterate current into place.....
② temp=(Node*)malloc(sizeof(Node));
  temp->data=>;
③ temp->prev=current;
④ temp->next=current->next
⑤ current->next->prev=temp
⑥ current->next=temp;
```



Removal Algorithm:

```
Node* current;
① //iterate current into place.....
②current->next->prev=current->prev;
③current->prev->next=current->next;
④free(current);
⑤current=NULL;
```



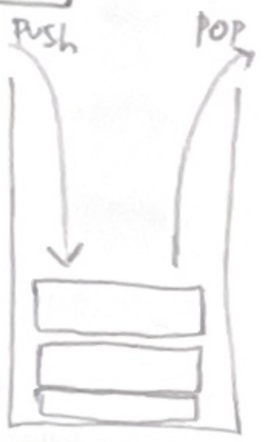
worst case → Big

Byte Size

char → 1 byte
int → 4 bytes
address → 4 bytes
double → 8 bytes

Abstract Data Types

Stack



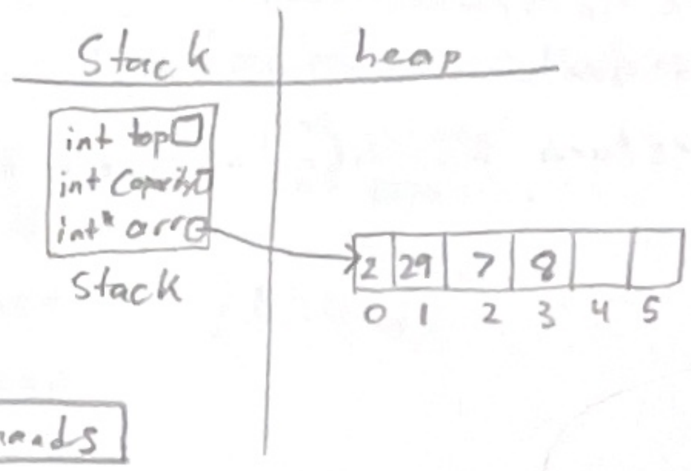
- Can use arrays or Linked Lists

```
typedef struct stack {
  int top;
  int capacity;
  int* arr;
} Stack;
```

Operations:

- **push** insert an item at top of the stack
- **Pop** remove and return top item
- **Peek** return top item
- **isEmpty** does the stack contain any items

Structure



top, index of first free space
capacity_array size

Queue

- First in First out

```
typedef struct {
  int front;
  int num;
  int capacity;
  int* arr;
} Queue;
```

Commands

Enqueue → inserts an item to the back of the queue

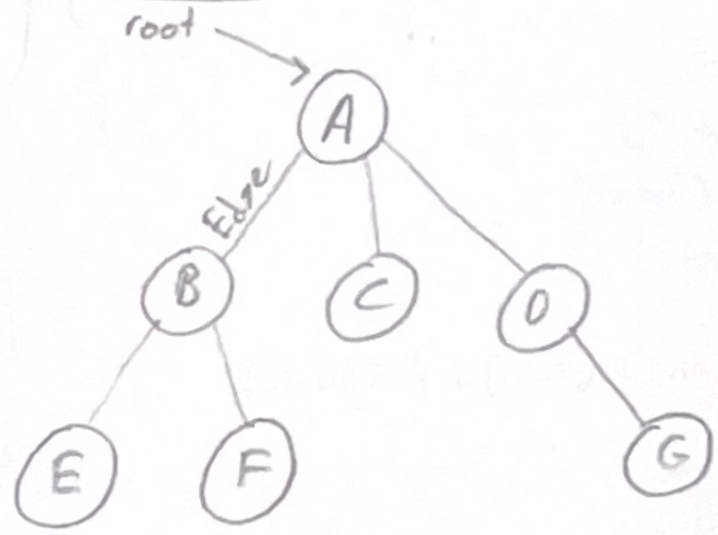
Dequeue → removes an item from the front of the queue

Circular array:

Back = (front + num) % capacity

front = (front + 1) % capacity

Tree (Data structure)



Traversals

Pre-order (T) Post-order

in-order

Terminology

Leaf → node with no children

degree → # of children a node has

Branching Factor → max degree any node can have

Perfect/Complete Binary Tree

- a perfect tree with h levels has $2^h - 1$ nodes

Criteria (Complete):

1. leaves are at most on two different heights
2. second to bottom level is completely filled
3. The leaves on the bottom are as far to the left as possible

BST Efficiency

1. Start at root
2. if key < node go left
3. if key > node go right

Recursion

minimum 2 cases:

1. Base case, exit
2. Recursive case, recursive call

Common uses

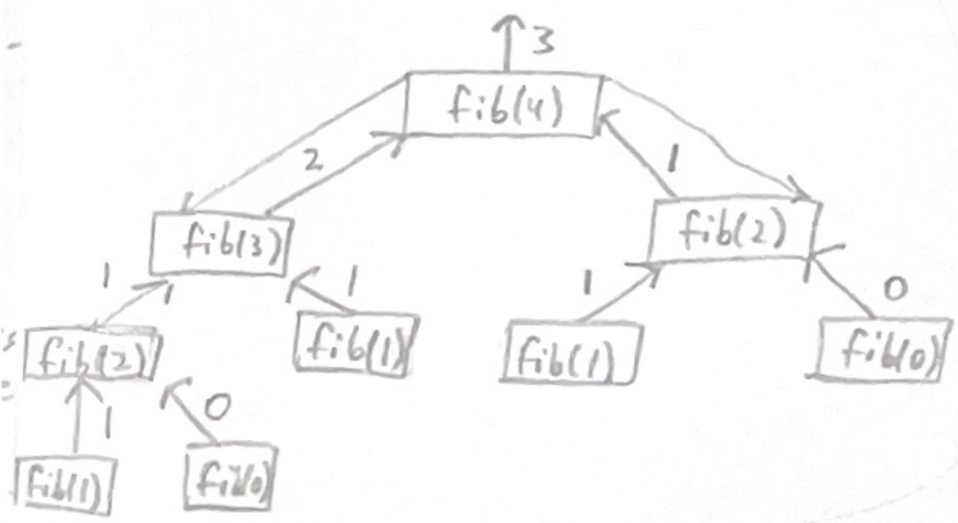
Fibonacci:

```
int fib(int n)
{
  if(n<=0)
    return 0;
  else if (n==1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Factorial:

```
int factorial(int n)
{
  if(n<=0)
    return 1;
  else
    return (n* factorial(n-1));
```

fib visualization:



BST search

```
int Search (BNode* nd, int key)
{
  if(nd==null) return FALSE;
  else if (nd->data==key) return TRUE;
  else
  {
    if(key< nd->data)
      return search (nd->left, key)
    else return search(nd->right, key);
  }
}
```

BST Insertion

```
BNode* insert (BNode* nd, int key)
{
  if(nd==Null); {
  BNode* newnode =(BNode*) malloc(sizeof(BNode));
  newnode->data=key;
  newnode->left=Null;
  newnode->right=Null;
  return newnode;
}
  else {
    if(key<nd->data)
      nd->left= insert(nd->left, key);
    else nd->right=insert(nd->right, key);
    return nd;
  }
}
```

Heaps

- Binary Tree with 2 properties:

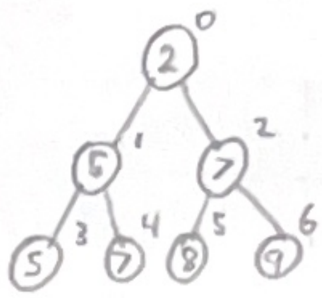
① Complete, All levels except the bottom are completely filled in and the bottom level has leaves as far to the left as possible

② Partially ordered, its either a max heap → every node has a larger value than its children or a min heap → every node has a smaller value than its children

Implementation

Option 1: Array

- Index nodes from top to bottom left to right. Completeness ensures no gaps in the array



2	5	7	5	7	8	9

Accessing

- At node i

① parent node → arr[(i-1)/2]

② Left child → arr[(2*i)+1]

③ Right child → arr[(2*i)+2]

Structure (MinHeap)

```
typedef struct MinHeap {
    int size;
    int capacity;
    int *arr;
} MinHeap;
```

Algorithms

Binary Search ⟶ $O(\log n)$

Sequential Search ⟶ $O(n)$

Tree Traversal ⟶ $O(n)$

Selection Sort ⟶ $O(n^2)$

Merge Sort ⟶ $O(n \log n)$

Insertion Sort ⟶ $O(n^2)$

Heap Sort ⟶ $O(n \log n)$

Hash Table

Handle Collisions

① Open addressing, look for a new array position. How:

① linear proping
② Quadratic probing
③ double Hashing

② Chaining.
Linked List or tree Structure

Linear Proping

-Just proceed forward to the next open space

Quadratic probing

-instead of proceeding one spot at a time, search spot, then spot +1
spot +4
spot +9
spot +16
⋮

Double Hashing

-Have a second hash function to determine the offset