Data Structures and Algorithms 8.27.24

Zybooks Reading 01:

* Data Structure: way of organizing, storing, and performing operations on data

Record → stores subitems (fields) each w/ an associated name

Array → stores ordered lists of items, accessible w/ an index

Linked List → stores an ordered list of items in nodes, where each node stores data and pointer to next node

Binary Tree → each node stores data and has up to 2 children (left and right)

Hash Table → stores unordered items by mapping (hashing) each item to a location in an array

Heap → max/min-heaps are trees where a node's key is greater/less than its childrens' keys

Graph → represents connections among items, where a vertex represents an item and an edge represents a connection

* Algorithm: sequence of steps to solve a computational problem or perform a calculation

longest common substring

binary search → searches a list (ex. array)

Dijkstra's shortest path → determines shortest path from a start vertex to each vertex in a graph

Efficient Algorithms: assessed by runtime and must increase no more than polynomially with respect to input size

NP-complete: set of problems where no efficient algorithm exists
- no efficient algorithm has been found
- no one has proven an efficient algorithm is impossible
- if an efficient algorithm exists for one NP-complete problem, it must exist for all ?

* by knowing problem is NP-complete, focus shifts to "good-enough" algorithms

* specific algorithms typically exist for specific data structures

Computational Complexity: amount of resources used by an algorithm
- runtime complexity: $T(N)$
  - $T \rightarrow$ # constant time operations
  - $N \rightarrow$ # inputs
  - best case $\rightarrow$ min N (cannot be 0)
  - worst case $\rightarrow$ max N
- space complexity: $S(N)$
  - $S \rightarrow$ # fixed-size memory units
  - $N \rightarrow$ # inputs
  - $k \rightarrow$ constant representing memory for loop-counters and list pointers
- auxiliary space complexity: space complexity without input data

Lecture 01 8.28.24

Ex. Detecting Duplicates

```
bool has_duplicates(a, n)
{
  for each element a[i] {
    for each element a[j] from i+1 to end {
      if a[i] == a[j]
        return true
  return false
}
```

runtime complexity: $\sim N^2$

aux-space complexity: K (doesn't depend on size of array since only needs to hold counter integers AFTER input is processed)

Faster Approach:

data = {3, 1, 3, 2}

seen = {f, f, f, f}

seen = {f, f, f, t}

seen = {f, t, f, t}

runtime complexity: $\sim N$

aux-space complexity: $\sim N$

What if values ranged from 0 to N^2?

space is $\sim N^2$

```
bool has_duplicates_fast(a, n) {
  allocate "seen" table
  for each element in a
    if seen[a[i]] == true
      return true
    seen[a[i]] = true
  return false
}
```

Zybooks Reading #2:   8.29.24

VS-Code Terminal:

Manage → Themes → Color Theme

Hamburger → Terminal → New Terminal

Debugging:

- Compile w/ -g flag
- click "Run and Debug"
- create a launch.json file
- click "Add Configuration"
- use gdb launch
- enter "${workspaceFolder}/[executable]" under "program"
- Add a breakpoint
- Run through debug window "start debugging
- Use "step over" to run program step-by-step
- Use "step into" to access function being called
- Press "stop" to exit debugger
- Make sure to exit debugger console/terminal and re-enter "bash" terminal
- can also watch certain variables in debug window
- can also use valgrind ./[executable] memory error detector

Lecture 02: 8.30.24

Compiling C Programs:

foo.h header file (text)

↓

foo.c source file (text)

↓

gcc -c compile (-c means to object files, not exe)

↓

foo.o object file (binary)

↓

gcc link (one or more object files)

↓

foo executable (binary, start at main)

Automating Compilation: make

consists of a set of rules for actions to compile targets from their sources

```
target : source1 source2
[tab] actions
```

Ex. compile hello.o from source hello.c

```
hello.o: hello.c hello.h
[tab] gcc -g -c hello.c
```

Ex. compile exe app from object files main.o and lib.o

```
app: main.o lib.o
[tab] gcc -o app main.o lib.o
```

Inside: make directed acyclic graph (DAG) data structure and performs topological sort algorithm

Duplicates File Structure:

dups.h → unit_dups.c, dups.c, time_dups.c

unit_dups.c → gcc -c → unit_dups.o

dups.c → gcc -c → dups.o

time_dups.c → gcc -c → time_dups.o

unit_dups.o, dups.o → gcc → unit_dups

dups.o, time_dups.o → gcc → time_dups

Assert: assert(condition)

- keeps executing if condition is true
- exits program w/ error code and prints msg w/ line # if condition is false

* we will write <u>unit tests</u> for every function using assert statements

* use #include <assert.h>

* return code 0 for success

* -Wall gives all warnings

* chmod +x [SHELL exe].sh
  to run shell script

Reading 03: 9.1.24

2.1 Pointer Basics:

pointer: var that holds another var's address
- has a data-type which determines what kind of address it holds

reference operator (&): obtains a variable's address

declaration: int *p = #

dereference operator (*): retrieves the data where the pointer is pointing

Null pointer: a pointer that points to nothing
- * 0 on most systems b/c 0 is not an address to anything

* be careful when declaring pointers and using the reference / dereference operators
  - → dereferencing a null pointer causes a runtime error
  - → dereferencing an unknown address causes a runtime error

2.2 malloc and free functions

#include <stdlib.h>

malloc (bytes); // basic use

malloc (sizeof (datatype)); // common form

void pointer: "universal remote" malloc() returns a void pointer
- → must be typecasted
  - ex. p = (datatype *) malloc (sizeof (datatype));

free (pointervar); // de-allocates memory used by a pointer
- CANNOT dereference a pointer after it has been free()'d - returns error
- calling free() for a pointer that hasn't been malloc() also returns an error

2.3 Memory Leaks

Memory leak: when a program loses track/access to previously allocated memory

→ failure to free()

garbage collection: some languages (ex. Java) automatically find these unaccessible memory locations and free's them

→ not C!

2.4 String functions with pointers

#include <string.h>

* arguments passed as char* can be modified by string.h functions but const char* cannot

ex. strcmp(), strcpy()

C string search functions:

strchr(), strrchr(), strstr()

2.5 The malloc function for arrays and strings

* normally, arrays and strings must be statically allocated
* malloc() can create dynamically allocated arrays

ex. pointer=(datatype*)malloc(numElements*sizeof(datatype));

ex. concatenating strings dynamically

p=(char*)malloc((strlen(str)+8)*sizeof(char));

* remember '\0' appended to ALL c-strings in these examples! *

Lecture 03: 9.2.24

Shifting:

2<<15 // 2 shifted 15 bits to the right

$= 2^{16} = 32768$

High/Low Level Programming Language:

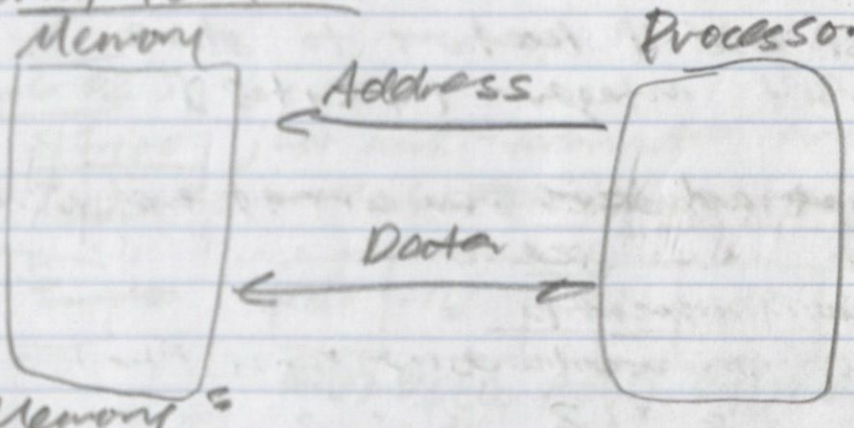
Low-Level → High-Level

Assembly — "as low as you can get before binary"

C

Python — "very abstracted"

Why Pointers?



Memory:

- 64 bits of address (18,446,744,073,709,551,616)
- each address stores 8 bits / 1 byte of data

Processor:

- load/store data in memory
- perform arithmetic/logic operations on data
- figure address for next instruction

Software:

- system: esp memory management
- numeric: fast matrix mult / graphics
- symbolic: data structures, circle and arrow diagrams

* Python was written in C

Memory: base 16 : 0-9 , a-f

(16 bit system)

address (pointer)	byte of data	variable name
0x~~ffff ffff ffff ffff~~		
0x~~ffff ffff ffff fffe~~		
...		
0x0000 7ffe f146 2085	'A', 0x41, 01000001	char c
...		
0x0000 0000 0000 0002		
0x0000 0000 0000 0001		
0x0000 0000 0000 0000		

* you must use 4 locations to store a 32-bit integer (4 bytes)

C Data Types and Sizes:

data type:	size:
bool	1
char	1
short	2
int	4
float	4
double	8
char *	8
int *	8

Ex.
declaration: int a
expression: a
evaluates to: value of a

declaration: int A[4]
expression: A
evaluates to: address of A[0]

Indexing Arrays and Pointer Arithmetic:

What happens when you add 1 to a pointer?
* increments by the size of the thing it points to

Lab 02: 9.3.24

Strings: just null-terminated arrays of chars
char *s = "abcde"

S → ← p

'a'	'b'	'c'	'd'	'e'	'\0'
&s[0]	&s[1]	&s[2]	&s[3]	&s[4]	&s[5]
s	s+1	s+2	s+3	s+4	s+5

* p-s is the length of the array

Pointer Variables and Arrays: moving away from numeric addresses and towards arrays and objects

p = &x; // p points to x
y = *p; // y gets value pointed to by p
q = p; // q gets same pointer as p
they point to the same thing

Reading 04 : 9.3.24

%lx // prints in long hex format

Iterating through strings w/ pointers:

```
for (int i=0; *(s+i); i++) {
    printf("%c", *(s+i));
}
```

2.8:

* malloc() is commonly used for pointers w/ structs

ex:

```
typedef struct myItem {
    int num1;
    int num2;
} myItem;

myItem* myItemP = NULL;
myItemP = (myItem*) malloc(sizeof(myItem));
```

Member Access Operator: ->

structPtr->memberName ≃ (*structPtr).memberName

2.9 Memory regions: Heap/Stack

4-regions of program memory:

Code - where program instruction is stored

Static memory - where global variables and static local variables are stored. They are allocated once and remain in the same location for the entirety of execution

The Stack - where local variables are stored. LIFO and automatically allocated and deleted

The Heap - where malloc() and free() allocate memory. Also called the free store - programmers have explicit control over allocation/deallocation

Lecture 04: 09.04.24

Dynamic Memory Allocation:

void *malloc(nBytes)
- allocates uninitialized memory

void *calloc(nObjects, bytesPerObject)
- initializes all values to 0

free(void *)

* valgrind --leak-check=full ./executable

Memory Management: most important aspect of computer system design

Regions of Memory:

Stack grows downward	function local variables
Heap grows upward	malloc(), calloc(), free()
Data fixed size	global and static vars, string constants "abc"
Code fixed size	program machine code

__Ex.__

```
double G = 3.14;
int main(int argc, char *argv[]) {
  int x = 1;
  int a[] = {4, 6, 6, 3, 7};
  char *s = "string";
  int *p = malloc(10*sizeof(int));
  static int t = 2;
}
```

__stack__: &x, a, &s, &p
__heap__: p * on midterm *
__data__: s, &t, &G
__code__: main

__Stack Frame__:

main	main	main	main	main	main	main
	f_1	f1	f1	f1	f1	
		f2	f2	f2		
			f3			
			}			

stack pointer
(bottom of stack

Lecture 05: 9.6.24

Array of char vs. String constants:

```
char a[] = "cat";    4 bytes
char *p = "dog";     4/8 bytes
                     x32 / x64
```

a: in stack, can change values but not reassign

'c' 'a' 't' '\0'

&p: in stack, string constant in data
can reassign but can not change values

p ⟶ 'd' 'o' 'g' '\0'

I/O Printing strings:

```
char *string = "hello"
can print with:
    printf("%s\n", string);
    puts(string);   // much simpler
```

I/O Reading Input Line by Line:

use fgets() to read one line at a time into a buffer and then process:

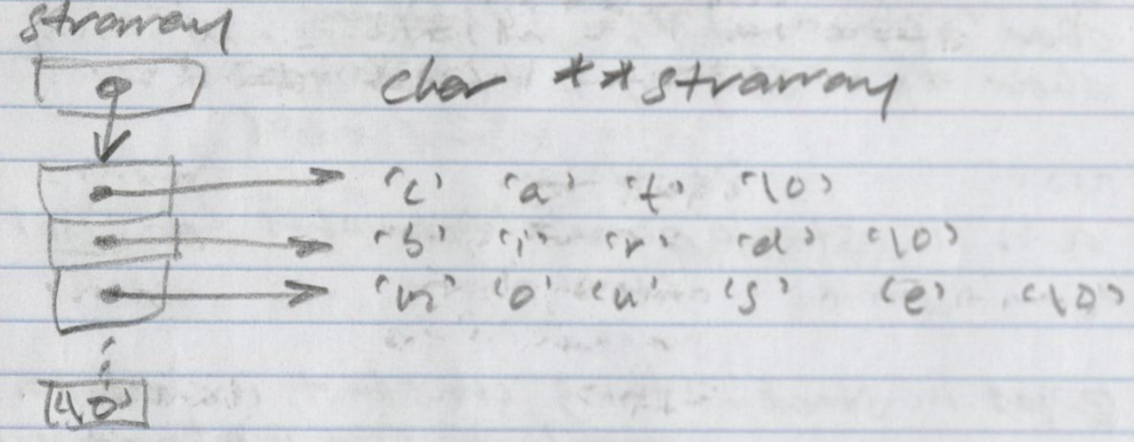
```
char buffer[BUFSIZ];
while (fgets(buffer, BUFSIZ, stdin)) {
    process(buffer)
}
```

I/O: Chomping Newlines:

- fgets() appends a '\n' to the end of each string
- we can chomp this by iterating through the buffer until reaching '\0' replacing any '\n' with '\0' on the way

Array of Strings:

char **strarray



Allocate and Free:

calloc(); // to allocate array of string pointers initialized to 0

strdup(); // to allocate individual strings and assign them to successive addresses in the array

free(); // loop over the string array to free each individual string
// free the string array itself

Reading 05 : 9/9/24

Abstract Data Types (ADTs) : data type described by predefined user operations, without particularly indicating how each operation functions

ADT	Description	Common DS
List	ordered data	array, linked lists
Dynamic Array	ordered data indexed access	array
Stack	FIFO	linked list
Queue	LIFO	linked list
Deque	items can be inserted and removed LIFO/FIFO	linked list
Bag	unordered, allows duplicates	array, linked list
Set	collection of distinct items	Binary search tree Hash Table
Priority Queue	each item has and is ordered by priority	Heap
Dictionary (Map)	maps keys with values	Hash table Binary search tree

* Allows for abstraction and optimization

The realloc function : realloc()

- reallocates a pointers memory block to a new size
- can be used to increase or decrease the size of dynamically allocated arrays

p = (type*) realloc(p, numElements * sizeof(type));

Vector ADT:

```c
#include "vector.h"
void vector_create(vector *v, int size);
void vector_destroy(vector *v);
void vector_resize(vector *v, int size);
int * vector_at(vector *v, int index);
int vector_size(vector *v);
void vector_push_back(vector *v, int value);
void vector_insert(vector *v, int index, int value);
void vector_erase(vector *v, int index);
```

Lecture Notes: 9.9.24

Struct Review:

```c
typedef struct {
    int x;
    int y;
} Point;
```

Mem Allocation:

```c
Point p; p.x = 1; p.y = 2;
```

	Label	Address	Value
		0xF	0
		0xE	0
int y		0xD	0
		0xC	2
		0xB	0
int x		0xA	0
		0x9	0
	p:	0x8	1

struct Point

Dynamic Array : data structure that can grow as more elements are added

Dynamic Array	
data	→ [0] [1] [2] [3] internal fixed array w/ data
capacity	number potential elements
size	actual number of elements used

* variant of vectors

Structure:

```
typedef struct {
    int *data;       // internal array
    int capacity;    // total # elements
    int size;        // # valid elements
} Array;
```

Regrowing: double the capacity when you need it

Methods: faking OOP with functions

```
array_create();  // create empty array
array_delete(Array *array);
array_append(Array *array, int value);
array_at(Array *array, int index);
array_index(Array *array, int value);
array_insert(Array *array, int index,
             int value);
```

Growing an Array : realloc()

```
void * realloc (*ptr, size_t size);
```

* leaves memory uninitialized
* will preserve values already there

Shifting Elements to the Right:

array_insert(array, 3, 34);

31 32 33 35 36

31 32 33 34 35 36

* use a loop *

from: index size

to: current index (or above)

operations: data [i] = data [i-1]

Computational Complexity:

Big O Notation: gives "order" of computation

O(1): constant, doesn't depend on # elements

O(n): proportional to # elements

O(n^2): proportional to square of ~~#~~ # of elements

Dynamic Array Complexity: Average(A), Worst(W)

Function	Time (A)	Time(W)	Space (A)	Space(W)
Append	O(1)	O(n)	O(1)	O(n)
At	O(1)	O(1)	O(1)	O(1)
Index	O(n)	O(n)	O(1)	O(1)
Insert	O(n)	O(n)	O(1)	O(n)

Pros / cons: Dynamic Arrays

Pros:
- grows as needed
- fast random access
- good cache locality

Cons:
- sometimes slow insert/delete
- worst case append (have more space allocated than you need)

Reading Notes: 9.10.24

Stack ADT: items are only inserted or removed from the top of the stack

push - inserts an item
pop - removes an item

LIFO ADT

Push (stack, x)
Pop (stack)
Peek (stack) // returns top of stack
IsEmpty (stack) // T/F if empty
GetLength (stack) // # items in stack

Queue ADT: items are inserted at the end of the queue and removed from the front

dequeue - removes and returns front item
enqueue - inserts item at end

FIFO ADT
can be implemented as linked list or array

Enqueue (queue, x)
Dequeue (queue)
Peek (queue) // returns front value
IsEmpty (queue)
GetLength (queue)

Deque ADT = items can be inserted and removed from the front and the back

push-front (deque, x)
push-back (deque, x)
pop-front (deque)
pop-back (deque)
PeekFront (deque)
PeekBack (deque)
IsEmpty (deque)
GetLength (deque)

Set ADT = unordered collection of distinct elements. elements can only be added if they do not exist

* can have any or multiple datatypes, usually distinguished by:
  key value - primitive data value; unique identifier for element
* can remove element from set this way too
  ↳ key value

* can search for a subset in a set as well
  "a set X is a subset of set Y only if every element of X is also an element of Y"

Set operations

Union (∪): $\{54, 19, 75\} \cup \{75, 12\} = \{12, 19, 54, 75\}$

Intersection (∩): $\{54, 19, 75\} \cap \{75, 12\} = \{75\}$

Difference (\\): $\{54, 19, 75\} \setminus \{75, 12\} = \{54, 19\}$

$X \cup Y$: every element from sets X and Y, no additional elements

$X \cap Y$: every element from both sets X and Y, no additional elements

$X \setminus Y$: every element in X but not in Y, no additional elements

* $\cup$ and $\cap$ are communicative, $\setminus$ is not *

filter: produce a subset of X that satisfies a particular condition

map: operation on a set X that produces a new set by applying a function F to that set

Static and Dynamic Set Operations:

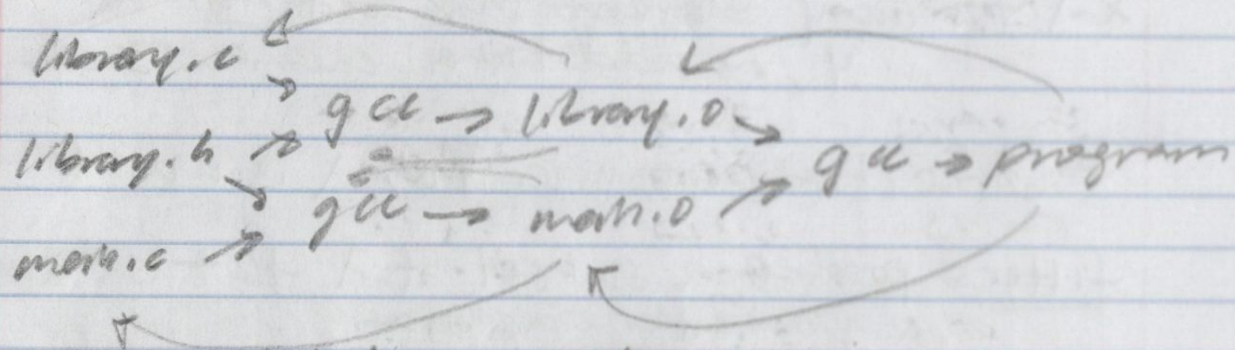
dynamic set: set that can be changed after construction

static set: cannot be changed after construction

Lecture Notes: 9.11.24

*realloc() does not always reallocate the memory at the same location

Make Motivation: automates the compiling process



nodes are linked by dependencies
domain specific language for DAGs?

Variables:
$(variable)

Rules:
TARGET: SOURCE
    COMMAND

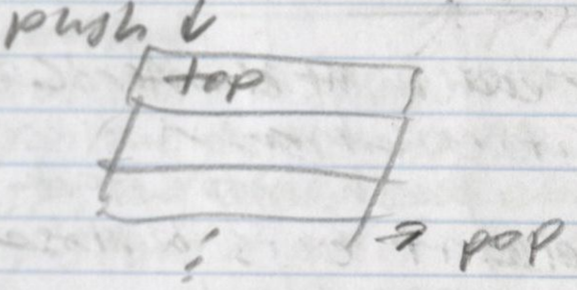
Macros:
$(wildcard *.c)
$(shell ls *.c)

Copy a Substring: strndup()

char *str = a b , c d e , f \0

char *substr = strndup(str+3, 3)
= c d e \0

Stack: ADT with 4 basic operations
push()
pop()
top()
empty()



LIFO

Function	Time	Space
push()	O(1)	O(1)
pop()	O(1)	O(1)
top()	O(1)	O(1)
empty()	O(1)	O(1)

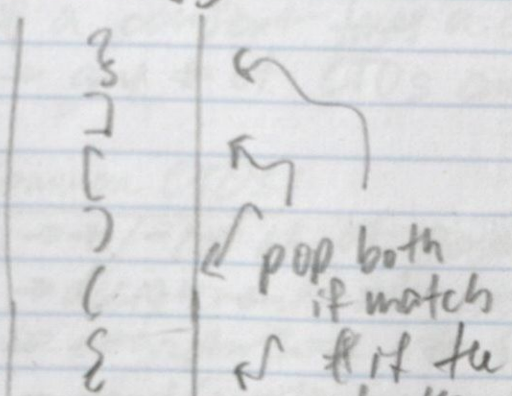
* You CAN technically insert/remove from any position but it would no longer be a stack *
→ just an array

Programming Challenge: PBB Matching

()	)(	[(){}]
✓	✗	✓

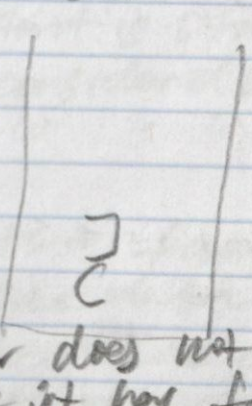
* parenthesis brace bracket matching *
* solutions involves use of the STACK *

ex. {()[]}



if the closer does not match the top of the stack it has failed

ex. (]



*if the stack is not empty it has failed

PBBMatch Pseudocode:

for each input character c
  if c is opener
    push onto stack
  else if c is a closer that matches top of stack
    pop stack
  else
    no match
    free stack
    must free stack before returning anything
if stack is empty
  match
  free stack
else
  not match
  free stack

* do these in Python
* lots of inputs and functions to automate tests

Reading07: 9.15.24

4.1. Searching and Algorithms

algorithm: sequence of steps to solving a task

Linear search: algorithm that starts from the beginning of a list, and checks each element until the search key is found or end of list is reached

$O(n) = n$

4.2. Binary search: search algorithm for sorted lists

Checks middle item first → if search key > middle value → do linear search on upper half vice versa.

* can be recursed *

↳ where its power comes from

* much faster than linear search

$O(n) = \log_2 n$

4.3. Constant time operations: an operation that, for a given processor, always operates in the same amount of time, regardless of input values

* since different processors will compute at different speeds

* a constant times a constant is still a constant

→ any # of CTOs can be considered as 1 CTO

Common CTOs:

→ +/-/*/ / of fixed ints or floats

→ assignment of fixed size data

→ comparison of two fixed size data values

→ read/write array element at particular index

4.4. O notation

Big O notation: mathematical way of describing how runtime of an algorithm behaves in relation to input size

Rules:

1. if $f(N)$ is a sum of several terms, $O(N)$ keeps only the highest order term
2. if $f(N)$ is a product of several terms, all constants are omitted

Ex. Algorithm steps: $5 + 13 \cdot N + 7 \cdot N^2$

$O(5 + 13N + 7N^2) = O(7N^2) = O(N^2)$

Composite Rules:

$c \cdot O(f(N)) = O(f(N))$

$c + O(f(N)) = O(f(N))$

$g(N) \cdot O(f(N)) = O(g(N) \cdot f(N))$

$g(N) + O(f(N)) = O(g(N) + f(N))$

* converting from one log base to another involves a constant term, which can be simplified in big O notation

$$\log_b(a) = \frac{\log_c(b)}{\log_c(a)}$$

c = new log base

$O(1) \to$ constant
$O(\log N) \to$ logarithmic
$O(N) \to$ linear
$O(N \log N) \to$ linearithmic
$O(N^2) \to$ quadratic
$O(c^N) \to$ exponential

4.5. Algorithm analysis
* commonly focuses on worst-case complexity *

worst-case runtime = runtime complexity for an input that results in longest execution

nested loops = multiply runtime of inner-loop w/ runtime of outer loop

Ex. Summation of consecutive numbers

$(N-1) + (N-2) + \dots + 2 + 1 = \frac{N(N-1)}{2} = O(N^2)$

Lecture Notes: 9.16.24

cntr-D signals end of stdin
./exe < file.txt reads in file from stdin

* creative use of data structures lowers degrees of freedom and makes problem solving easier and more elegant

staque
queue
deque
set

all restricted versions of dynamic arrays

Staque: LIFO
"stack of plates"

Queue: FIFO
"waiting in a queue/line"

Analysis

Function	Time (A)	Space (A)
Push	O(1)	O(1)
Pop	O(n)	O(1)
Front	O(1)	O(1)
Empty	O(1)	O(1)

Deque: LIFO and FIFO
"double ended queue"

* can build these ADTs on other data structures as well

Set: unique and unordered groupings

add (value) // add value to set
contains (value) // check if value in set
remove (value) // remove value in set (if it exists)

* ordering sets is very important to optimize sets (think contains())

Implementations:
- store values in array in order received
- ensure values are unique
- use linear search (for now)

Functions	Time(A)	Space(A)
Add	O(n)	O(1)
Contains	O(n)	O(1)
Remove	O(n)	O(1)

O(2n) → O(n)

Constant Time Operations: always takes the same amount of time regardless of input values

ex.
- arithmetic operations on fixed size vars
- assigning scalar variables
- loops with fixed number of iterations
- index into an array
- comparisons

Big O Notation: given f(N) as the exe time
O(N) is the order of growth rate of f(N)

- What is N?
  - → number of data items, size of something
- Ignore?
  - → coefficients
- Sum of terms?
  - → take the highest order term
- Product of terms?
  - → product of orders

* important concept, on exams *

Classic Big O Complexity:

- O(1)
- O(N)
- $O(N^2)$
- O(log N)
- O(N log N)
- $O(C^N)$
- O(N!)

O(1): constant time operations only
→ array lookup and compare
→ if (A[i] == key) ...

O(N): time proportional to N
→ linear search
→ for (i = 0; i < SIZE; i++)
    if (A[i] == key)
      return i

$O(N^2)$ and other polynomials:
→ nested loops
ex. for each i in N
for each j in N
...
ex. naive finding duplicates
constructing multiplication table

$O(\log N)$: divide-and-conquer, each
subproblem is constant time
→ binary search
→ range cuts in half with each iteration

$O(N \log N)$: divide-and-conquer, each
subproblem is $O(N)$
→ often an optimization of $O(N^2)$
→ various sorting algorithms
(merge sort)

$O(c^N)$, $O(N!)$: exponential problems
complexity grows geometrically w/ size
→ breaking a password of length N
by trying all combinations

Ex. Dot Product
$O(N)$

Ex. Matrix Multiplication
$O(N^3)$

Ex. Euclidean GCD
$O(\log N)$

Lecture Notes: 9.17.24

Palindromic Permutations

- words that can be scrambled into a palindrome
- use a set

1. enter in set
2. check if in set
3. remove if in set
4. if only one item left in set at end then it is a palendrome

RPN Calculator:

- enter inputs and operators line at a time
- tokenize into a string with a space b/w each line

3
4 → "3␣4␣+"
+

& strtok() helps with this
- use char delim = [" "];

Reading Notes: 9.18.24

Recursive Algorithm: an algorithm that breaks a problem into subproblems and applies itself to solve those subproblems.

Base Case: A case where the recursive alg. stops w/o applying itself to another subproblem

Recursive Function: A f'n that calls itself. Often used in recursive algorithms

*Binary Search is often implemented recursively

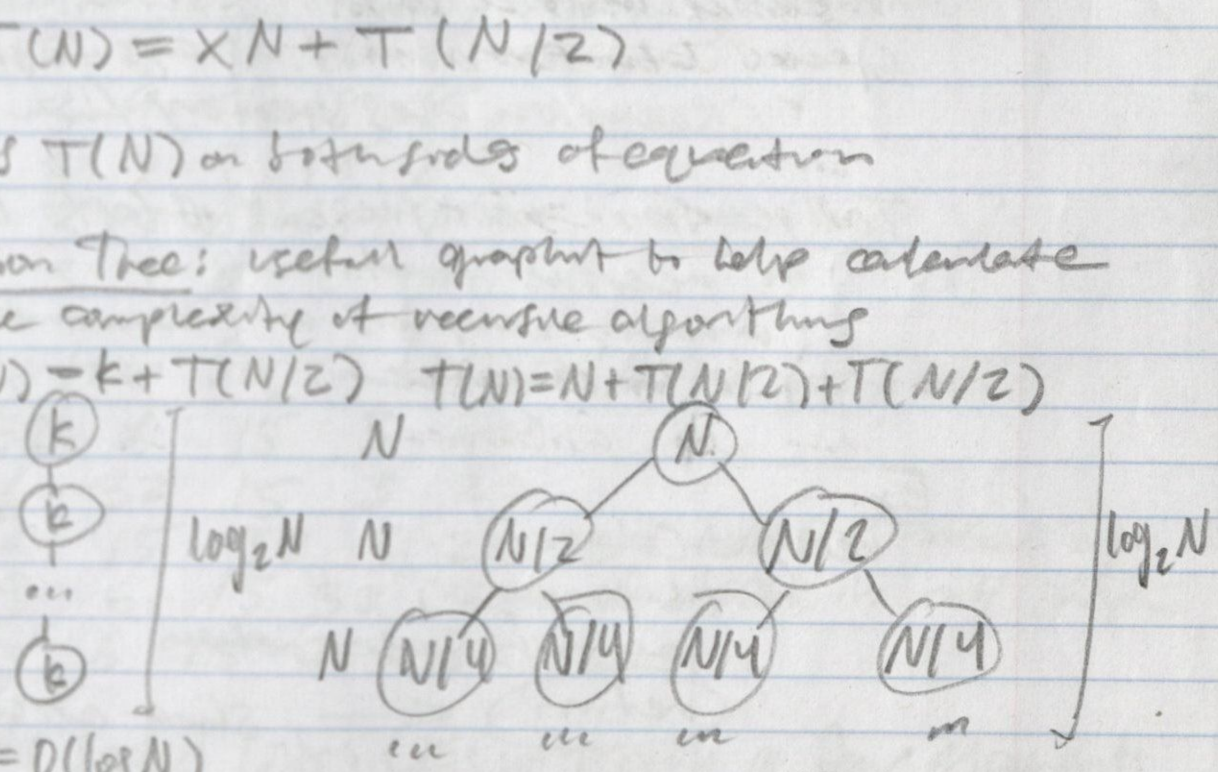
Recurrence Relations: used to calculate the time complexity of recursive algorithms

$$T(N) = xN + T(N/2)$$

* has $T(N)$ on both sides of equation

Recursion Tree: useful graphic to help calculate time complexity of recursive algorithms

$T(N) = k + T(N/2)$ $T(N) = N + T(N/2) + T(N/2)$



$T(N) = O(\log N)$ $T(N) = O(N \log N)$

Lecture Notes: 9.18.24

2^{10} - KB
2^{20} - MB
2^{30} - GB
2^{40} - TB

Linear congruential random number generator

Recursion:
  Why?
  - Because we can
  - Insights into problem solving
  - Sometimes the only reasonable solution
  - But you can go overboard
    - There are costs!
    • (Iterative almost always faster)
    • (Recursion uses memory from the stack)

Tail recursion: recursive call is last line
  in recursive function
  • modern compilers will convert these
    into iterative

Ex.
```
recursive(data)
    if (base case)
        perform base action
        return      <- short circuit return

    return not-base action
    recursive(new data)
```

Variants:
- Return data?
- forward or backward?
- More than one base case?
- Multiple recursive calls?
  - ← with trees

* Examples in Zybooks and Slides "LOSTS"
&
Section 4.10

Reading Notes: 9.19.24

Sorting: process of converting a list into ascending or descending order

Selection Sort: $f(N) = (N-1) \cdot \frac{N}{2} \Rightarrow O(N^2)$

7 9 3 18 8
3 9 7 18 8
3 7 9 18 8
3 7 8 18 9
3 7 8 9 18

Insertion Sort: $f(N) = (N-1) \cdot (\frac{N}{2}) \Rightarrow O(N^2)$

32 6 15 3 20
6 32 15 3 20
6 15 32 3 20 ← happens index by
3 6 15 32 20 ← index in all steps
3 6 15 20 32

Nearly Sorted Lists: Lists with only a few elements out of order
Insertion Sort $O(N) = N$ in these cases

Bubble Sort: $O(N^2)$
* considered impractical for real-world use as faster methods exist
* only swaps adjacent elements
* best and worst runtime complexity is $O(N^2)$

Radix Sort: sorting method only for integers
uses Buckets: a collection of integer values that all share a particular digit
ex. 57, 97, 77, 17
(share 7 in the one's place)
Time Complexity: $O(N)$
Space Complexity: $O(N)$

Steps:
- elements placed into buckets based on current digit value
- array result by removing elements from buckets in order
(lowest bucket → highest)
- repeat for all digits
(least significant → most)

RadixGetMaxLength(array, arraySize);
→ returns max number of digits in array
RadixGetLength(value)
→ returns number of digits

* uses another bucket to deal with positive values
(must reverse the array created here)

* often also done in base 2 (only 2 buckets needed)

Lecture Notes: 9.20.24

* Examples of linear and binary search in Zybooks 4.10

Recurrence Relations:

Factorial:

$$\left.\begin{array}{c} k \\ \vdots \\ k \\ \vdots \\ k \\ \vdots \end{array}\right\} N$$

$T(N) = k + T(N-1)$

$O(N)$

Naive Duplicates:

$$\left.\begin{array}{c} N \\ \vdots \\ N-1 \\ \vdots \\ N-2 \\ \vdots \end{array}\right\} N$$

$T(N) = N + T(N-1)$

$O(N^2)$

*think $\sum_{0}^{\infty}$ *

Fibonacci:

$T(N) = 1 + T(N-1) + T(N-2)$

$O(2^N)$

1

N-1 N-2

N-2 N-3 N-3 N-4

N

Slow and Fast Sorting Algorithms:
Simpler slower ones: $O(N^2)$
- Bubble sort
- Insertion sort
- Selection sort

Fast, recursive ones: $O(N \log N)$
- Merge sort
- Quick sort
- Heap sort

$O(N^2)$ Algorithms: all take multiple linear passes through array moving elements from an unsorted to sorted region

- Selection Sort: selects smallest element from unsorted region and puts it at the end of the sorted
- Insertion Sort: take the current first element of the unsorted region and insert it in the right place in the sorted region
- Bubble Sort: lots of passes swapping pairs of adjacent elements that are out of order. very inefficient

*Study these — website on C10 slides very good resource

Adaptiveness and Stability :

Adaptive : take advantage of data that is already partially sorted
- bubble : yes
- insertion : yes
- selection : no

Stable : doesn't change order of elements that are already in sorted order. Important for multi-level sorting
- bubble : yes
- insertion : yes
- selection : no

Which is fastest?

operations roughly:

$$c[N+(N-1)+(N-2)+\ldots+1] = \frac{cN(N+1)}{2}$$

2 kinds of OPS :
- comparisons
- swaps or moves

* highly data and machine dependent

fastest to slowest :
1. insertion improved
2. selection
3. insertion
4. bubble
5. bubble improved

All:
Time : $O(n^2)$
Space : $O(1)$

Reading Notes: 9.23.24

5.7 : Merge Sort

sorting algorithm that divides a list into 2 halves, recursively sorts each half, and then merges sorted halves to produce a sorted list

- recursive partitioning continues until list is 1 element long, as that is already sorted
- uses 3 indices i, j, k
  - i : index of first element in list
  - j : divides list into 2 halves
  - k : index of last element
- i - j : left half
- j+1 - k : right half

Time : O(N log N)

Space : O(N)

5.8 : Quick Sort

sorting alg that recursively partitions input into low and high parts (both unsorted) and recursively sorts those parts

- pivot : can be any value in the array, typically the middle value
- pivot = array[midpoint-index]
- low / high partitions ≤ / ≥ pivot

Time : O(N log N)

worst case : O(N²)

Lecture Notes :

9.23.24

Key to getting (N log N): divide and conquer
- divide problem into subproblems
- solve subproblems recursively
- combine partial solutions

* subdivision is a log N process

Solving Suprocesses is a O(N) process:
- Merge sort: merge already sorted smaller arrays
- Quick sort: partition values around a pivot value according to < / > the pivot

* Results in O(N log N) algorithms

Merge Sort:

- Merging two lists is O(N)

2 9   6 7 8

2 6 7 8 9

- Divide and Conquer (O log N)

9 2 8 6 7

9 2   8 6 7

9   2   8   6 7

6   7

Combine:

9 2 8 6 7

2 9 6 7

6 7 8

2 6 7 8 9

- Merge left/right subsequences into an ordered sequence
- Requires $O(\log N)$ space on the stack (since algorithm is recursive)
- Requires $O(N)$ extra space to hold copy of data while merging

* Zybook and website in L11 slides

Analysis...

Time	Space
$O(n\log n)$	
$O(n\log N)$	

Quicksort:

partitioning elements < or > a pivot element
pivot can be anything, usually middle element

9 2 8 6 7

2 6 7 $\boxed{8}$ 9

✓ X X

i → i p j

2 7 1 6 5 8 3 4

swap

✓ i → i p j ← j

2 4 1 6 5 8 3 7

when i and j cross, list sorted
j always on or less than p

Divide and Conquer:

9 2 8 6 7

2 6 7 $\boxed{8}$ 9

2 $\boxed{6}$ 7

2 6 7 8 9

Weak Point: QS works terribly when
pivot is max or min values
- approaches O(N) recursive stages instead of O(logN)
- work not divided equally

* another good animation on L11 slides

Analysis

	Time	Space
B	$O(n\log n)$	$O(\log n)$
A	$O(n\log n)$	$O(\log n)$
W	$O(n^2)$	$O(n)$

Adaptive? No
Stable? No

Reading Notes: 9.24.24

6.1. List ADT

common ADT for holding ordered data

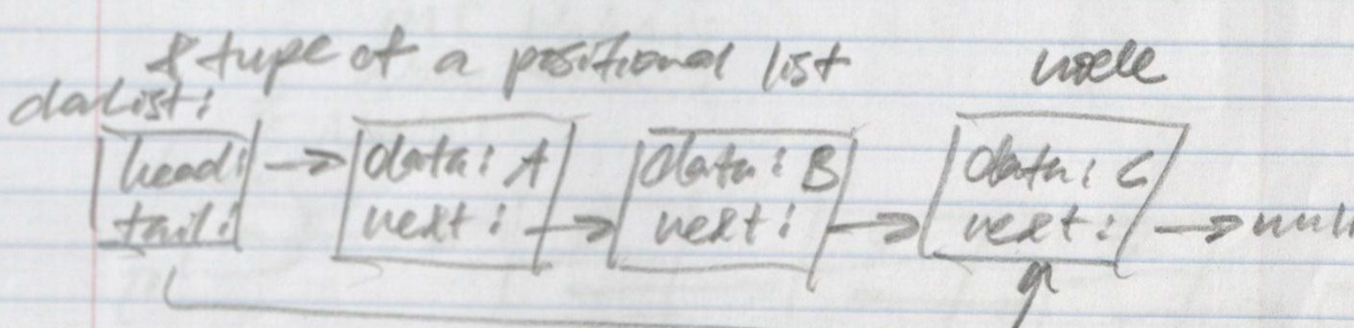
Append(list, x)
Prepend(list, x)
InsertAfter(list, w, x)
Remove(list, x)
Search(list, x)
Print(list)
PrintReverse(list)
Sort(list)
IsEmpty(list)
GetLength(list)

6.2. Singly-linked lists

- data structure for implementing a list ADT
- each node has data and a pointer to the next node

head: first node
tail: last node

& type of a positional list



Append: add node to end of linked list
Prepend: add new node before list's head node

Insert:

InsertAfter: inserts a new node after a pre-existing list node

Remove:

RemoveAfter: removes the node after the specified list node

- if current Node is null, RemoveAfter removes first node

LinkedList Search:

search algorithms return first node whose data matches that key

→ null if nothing found

→ starts at first node and searches through until item found

- linear search?

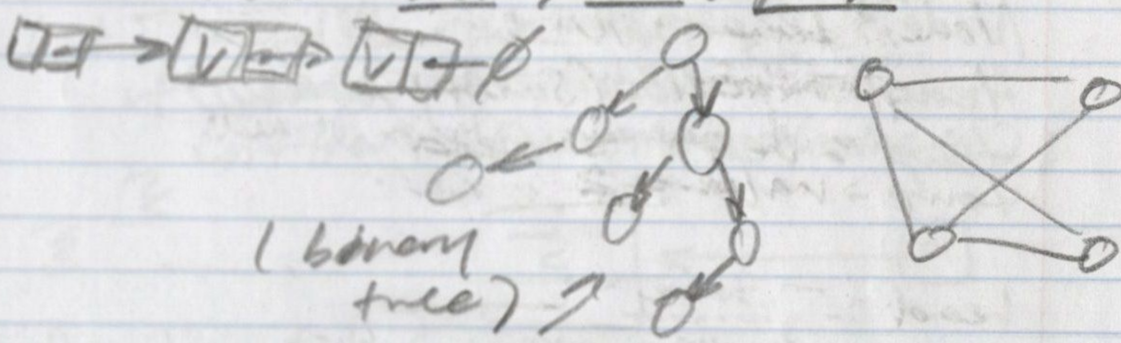
Lecture Notes: 9.25.24

Quicksort faster than Mergesort

But both 1000x faster than N^2 sorts

Linked Lists: type of dynamic data structure

2 styles of implementation: both dynamic
- Array BASED — dynamic arrays, hash-tables
- LINKED — list, tree, graph

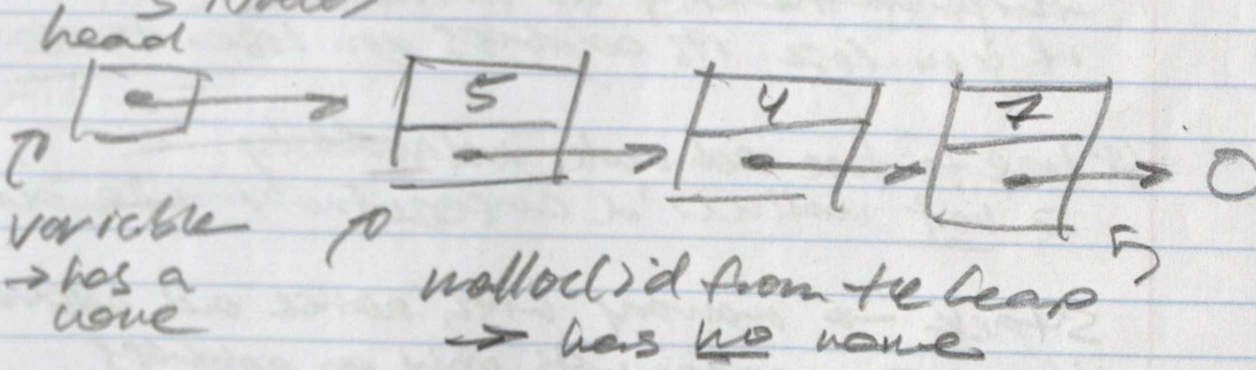


* all recursive structures *

ADT = Deques, Stacks, Queues, Sets, Lists
→ were been implementing w/ dynamic arrays
→ can implement them all w/ linked lists too

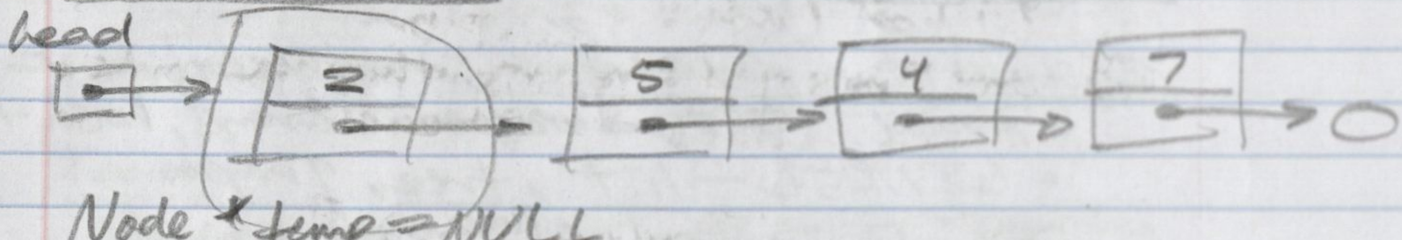
Node Definition:

```
typedef struct Node Node;
  struct Node {
       int value;
      Node *next;
  } Node;
```

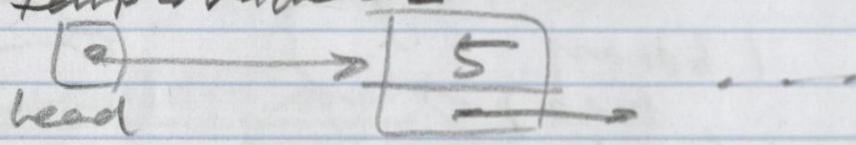


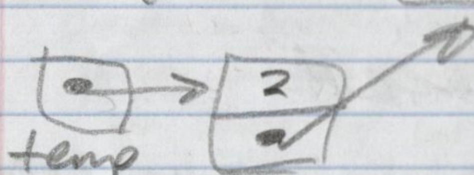
no name → data lost

Insert new Node: "at head"



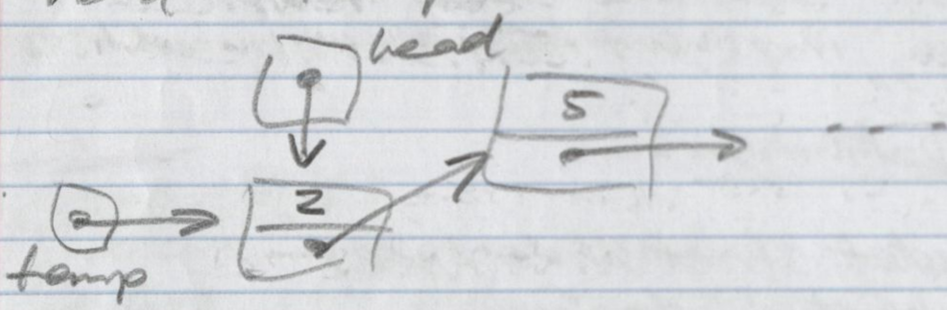

```
Node *temp = NULL
temp = malloc(sizeof(Node))
temp -> next = head
temp -> value = 2
```





How to Print 4:
```
printf("%d", head->next->next->value)
```
* pointer chasing

```
head = temp
```



* all nodes on the heap
* all variables have names
  things on the heap do not have names
  if you lose its address you lose the data

* have to free each node individually
  -> not malloc()'d consecutively like arrays

Stack -> memory with names and addresses
Heap -> memory with only an address

Reading Notes: 9.22.24

6.6. Linked Lists: Recursion

Forward Traversal: recurse through entire LL with node→next calls

```
List TraversalRecursive(node) {
   if (node not null) {
      ListTraversalRecursive(node→next);
   }
}
```

Searching: linearly and recursively searches through LL and returns first instance of key

Reverse Traversal: same logic as forward traverse, except visit node after recursive call
→ go backwards as recursive calls are pulled off the stack

6.7. Stacks using linked lists
- lists head node as top of stack

6.8. Queues using linked lists
- lists head points to front of queue
- lists tail points to end of queue

Lecture Notes: 9.27.24

* Exs of linked list methods in zyBooks 6.9
    node_create
    node_delete
    list_print_iterative
    list_delete_iterative
    list_add_after
    list_remove_after
    list_add_head
    list_remove_head

Reading Notes: 9.29.24

Doubly Linked-Lists: each node has
- data
- pointer to next node
- pointer to previous node

* each node has two pointers or "links" *
→ type of positional list

Appending:
to empty list:
- point head to new node
- point tail to new node

non-empty list:
- point tail → next to new node
- point new → previous to tail
- point tail to new node

Prepending:
```
if (list→head == null) {
  list→head = newNode;
  list→tail = newNode;
} else {
  newNode→next = list→head;
  list→head→prev = newNode;
  list→head = newNode
}
```

Inserting: ex on ZyBooks 6.11
* uses curNode and sucNode *
sucNode = curNode→next

Remove: ZyBooks 6.12

```
ListRemove(list, curNode) {
  sucNode = curNode -> next;
  predNode = curNode -> prev;
  if (sucNode != null) {
    sucNode -> prev = predNode;
  if (predNode != null) {
    predNode -> next = sucNode;
  if (curNode = list -> head) { //removed head
     list -> head = sucNode;
  if (curNode = list -> tail) { // removed tail
     list -> tail = predNode;
  }
```

Linked List Traversal: algorithm that visits and performs an operation on each node of a linked list

```
while (curNode != null) {
```

* doubly-linked-lists can do a reverse traversal

Searching algorithms for linked-lists
- → same algorithms
  - implemented differently
- → study these
- → different algs for single/doubly LL

Dummy Node: node with unused data always residing at the head (or tail)

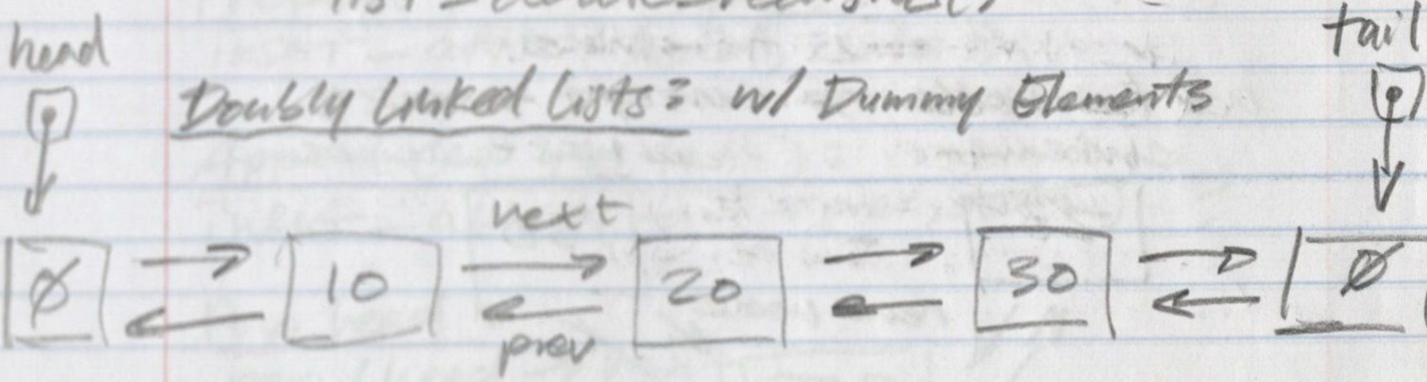
* lots of exs in ZyBooks 6.15

Lecture Notes: 9.30.24

+ more exs of recursive methods on linked lists in ZyBooks 6.9

```
list_print_recursive()
list_print_reversed_recursive()
list_delete_recursive()
```

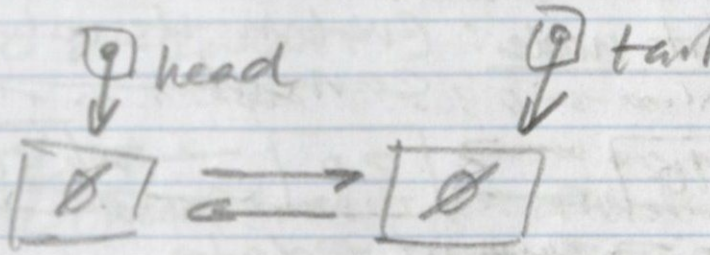
Doubly Linked Lists: w/ Dummy Elements



Why?

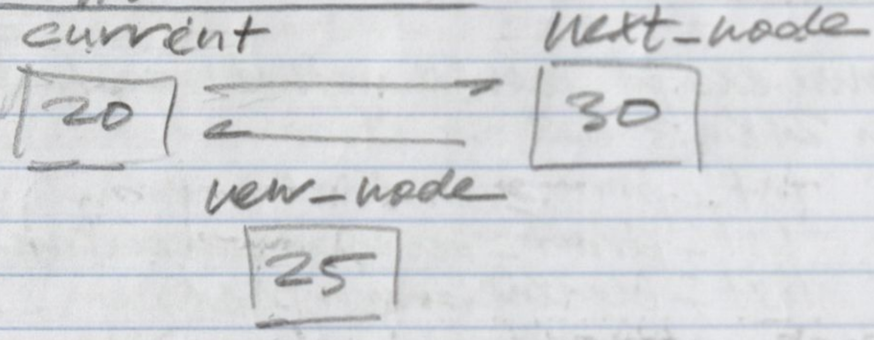
- eliminates many special cases
  - -> inserting always b/w 2 nodes
- solely to make programming easier

IsEmpty:

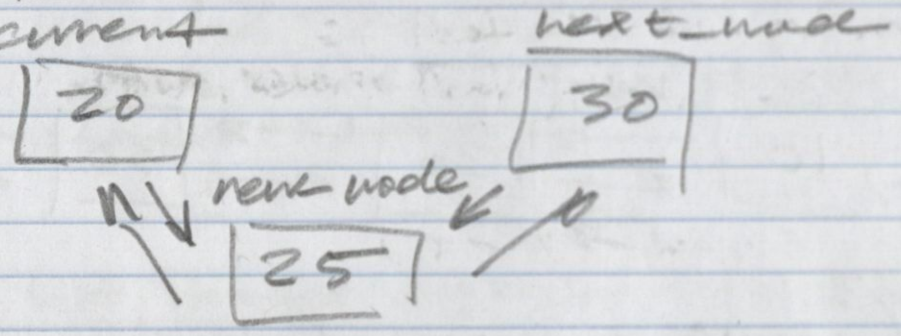



```
if (head->next == tail &&
    tail->prev == head)
```

Insert after current node:

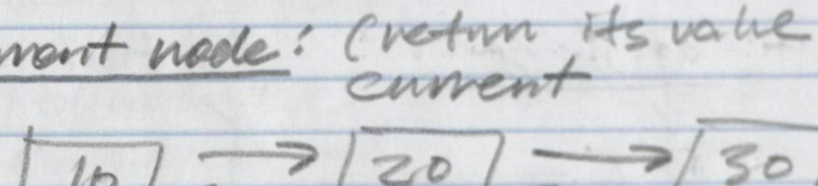


next_node = current->next

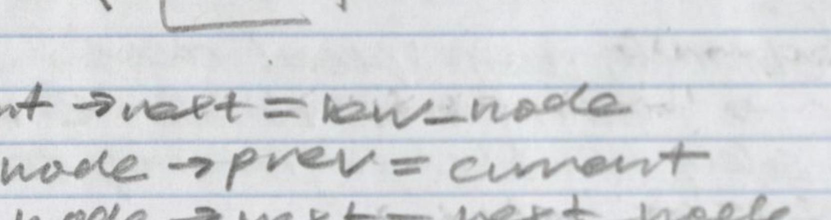


current->next = new_node
new_node->prev = current
new_node->next = next_node
next_node->prev = new_node

Pop current node: (return its value)

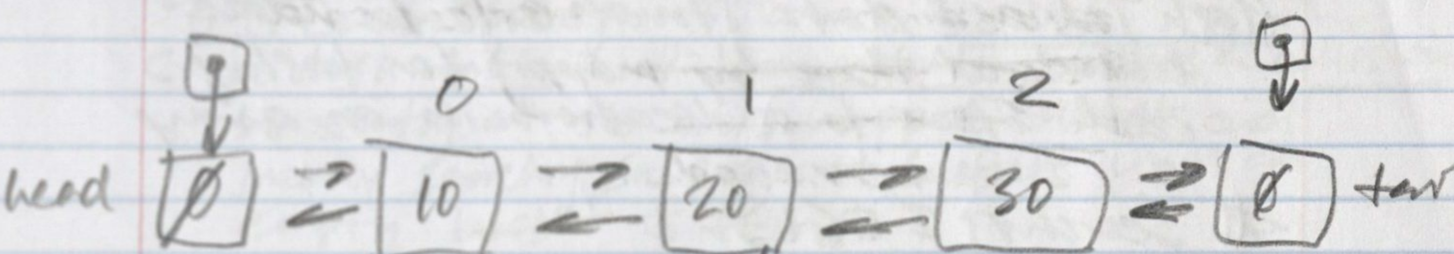


int data = current->data
current->prev->next = current->next
current->next->prev = current->prev



free(current)
return data

Data Structure:



Prepend (at the head):
insert_after(head, data)

Append (at the tail):
insert_after(tail->prev, data) ?

Pop head:
pop(head->next)

Pop tail:
pop(tail->prev)

Insert at Index:
- count to index and remove after

Delete at Index:

Print List:
while (head->next != dummy)

Reading Notes 10.1.24

Hash Tables: data structure that holds unordered items by mapping (or hashing) each item to a location in an array

** Searching: O(1)

Key: value used to map to an index

Bucket: hash-table array element

Hash Function: computes bucket index from item's key

* keys should be unique

Hash Table Operations:

common hash fn: modulo operator (%)

→ computes integer remainder

ex. for a 20 element hash table use key % 20 (maps to indices 0-19)

will map $\frac{\text{num\_keys}}{\text{num\_buckets}}$ keys to each bucket

Collision: when 2 keys are mapped to the same bucket

open-addressing: collision resolution technique

Chaining: handles hash table collisions by using a list for each bucket

* has all normal functions like insert, remove, search, etc...

Reading Notes 3 10.3.24

7.4. Linear Probing

a hash table w/ linear probing handles a collision by starting at the key's mapped bucket and linearly searching subsequent buckets until an empty bucket is found

Empty Bucket types:

empty-since-start: bucket that has been empty since the hash table was created

empty-after-removal: bucket that had an item removed that caused the bucket to be empty

Insert w/ Linear Probing:

1. uses item's key to determine initial bucket
2. linearly probes each bucket
3. inserts at next empty bucket

Remove w/ Linear Probing:

1. use key to get initial bucket
2. linear probe until:
   a) matching item found
   b) empty-since-start bucket found
   c) all buckets probed
3. if found, remove and mark empty-after-removal

Search w/ Linear Probing:

1. use the sought item's key to get initial bucket
2. linear probe until
   a) matching item found (return)
   b) empty-since-start found (return null)
   c) all buckets probed (return null)

7.5. Hash table resizing

resize operation – increases # of buckets, typically by:
next prime # ≥ N × 2

complexity: O(N)
must reinsert all items from old array into new one

load factor: # items in hash table / # buckets

Resize Thresholds:
- Load factor
- open addressing: # collisions during an insert
- chaining: size of bucket's linked list

7.6. Common Hash Functions

* a good hash function minimizes collisions *

perfect hash function: maps with 0 collisions
search, insert, and remove all have **O(1)** runtime

Modulo Hash Function: key % N

Mid-square HF: squares the key, extracts R digits from the middle of the result, and returns that % N
- For N buckets, $R \geq \lceil \log_{10} N \rceil$

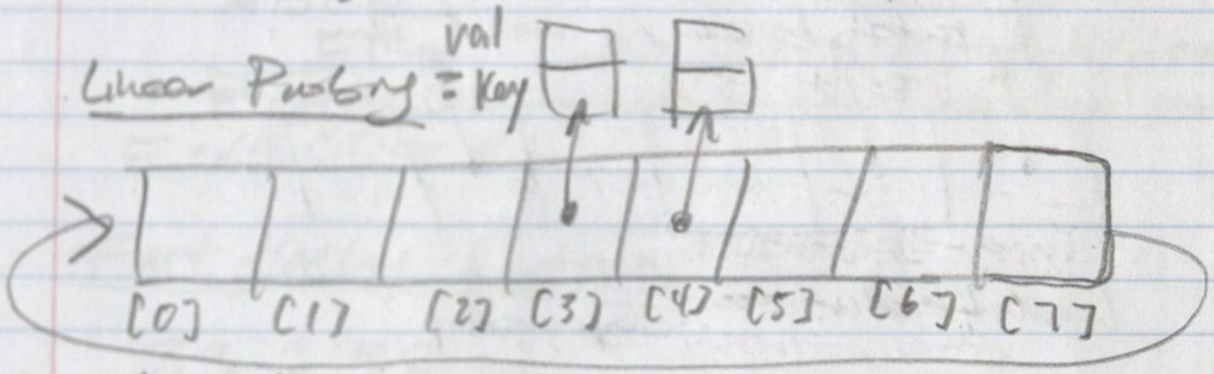
Mid-Square Base 2 HF: above usually done in binary to simplify extracting substring from string
$R \geq \lceil \log_2 N \rceil$

Multiplicative String HF: repeatedly multiplies hash value and adds ASCII value of each char in the string. Returns remainder of sum % N

Lecture Notes: 10.4.24

Ex of database using a chaining hash table
Zybooks 7.3K
→ has all hash-table methods

Linear Probing:



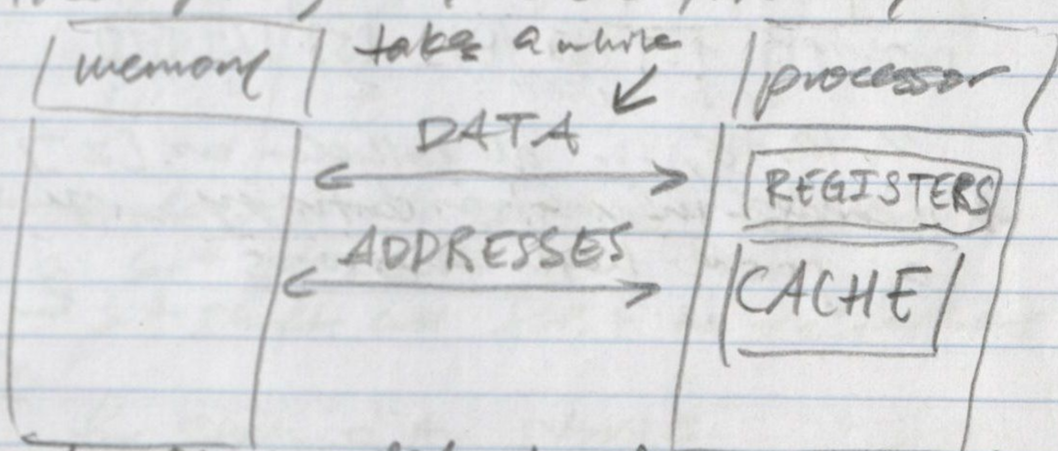
hash(key) % num_buckets

* just puts the node at the first available bucket
* next homework on this

- Python dictionaries use these internally
- Linear probing used more than chaining
  - quadratic probing

Caching:

- linear probing very cache friendly

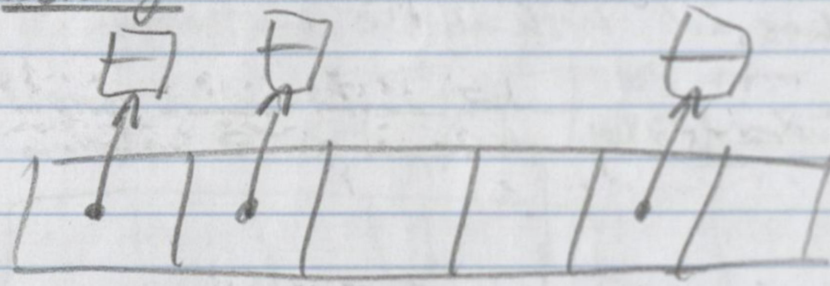


* try to optimize data transfer
  temporal locality
  Spacial locality

Lecture Notes: 10.7.24

More on Hash Tables

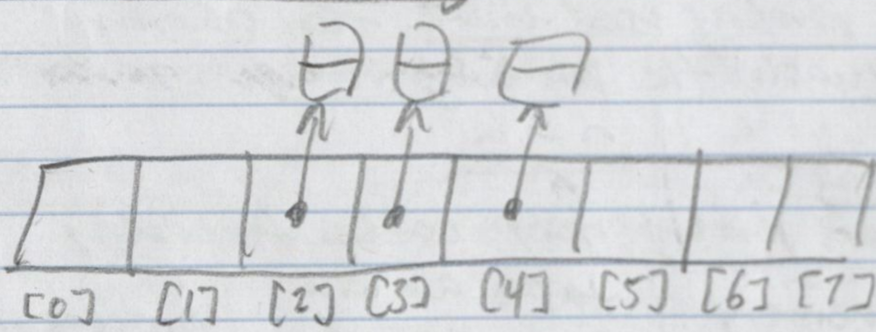
Resizing:



load factor = $\frac{\text{size}}{\text{capacity}}$

α: maximum allowable load factor (0.5)
- double the capacity
  - calloc() new array
  - copy over values (re-insert)

Collisions and Hashing:



2, 10, 18, ... all collide on [2]

* increase information entropy with a good hash functions

Hash Functions and Multiplication:

- multiply key by large (often prime) number
- extract bits from product

```
   ABCD
     EF
  ------
  F·ABCD
E·ABCD<<
```

this shift makes high/med bits a good random number

Fast Hashing: Shift and Add:

33 = 32 + 1

→ (shift left 5) + 1

in base 2 (binary)

* we do this because multiplication and division are slow

XOR:

(sum w/o carry)

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

* we do this b/c addition is also a bit slow *

→ bit shifts and XORs are the fastest

Ex.

```
for each c in string:
    hash = ((hash << 5L) + hash) + c;
    // hash = 33 * hash + c
```

How many buckets?

- ZyBooks says use primes
  - → not in practice
- Power of 2 buckets
- Double to resize
- Don't need %
  - → use bitmask

Ex. mid-square vs Knuth's multiplicative hash functions

Python:

real OOP language

→ has classes with methods

• instead of "fake" methods in C

Built in Datatypes:

• no types

• type(var) to see type

Lists: [] # dynamic array

Sets: {} # set ADT

Tuple: ( ) # immutable list

*everything in Python is an object of a type and has corresponding methods

dir(type) # lists all methods corresponding to type

regular methods: object.method()

magic methods: __method__

• can be used normally msg.__contains__('r')

→ automatically compiled to keywords

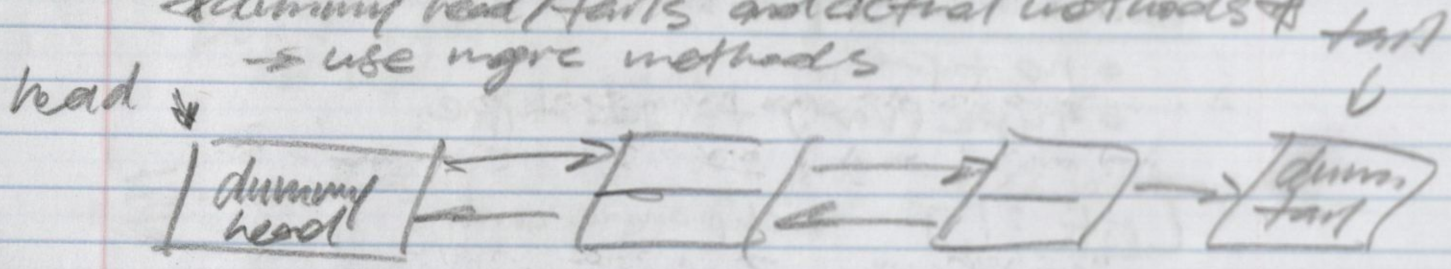
__contains__ → in

Lecture Notes: 10.11.24

* went over Python debugging and unit test *
import unittest

Doubly Linked List Class:

* dummy head/tails and actual methods *
→ use magic methods



Parsing data from stdin

- read string from stdin
- use split() to convert string into list of strings
- use list comprehensions to convert list of strings to list of ints

* use ', '.join(list) puts list of strings back into one string *

Python Doubly Linked List Class:

Use Python List Methods:

- append()
- clear()
- __str__
- __bool__
- __iter__
- __next__
- __contains__

def __init__(self, ...)
↳ referring to its own data

"constructor"
internally initializes class elements

* examples of all of this in ZyBooks 8.7 *

* automatic garbage collection *
→ don't have to malloc() or free() anything

__method__ → magic methods
method → internally used methods
.method → externally used methods

for _ in range(list):
↳ used if don't need this variable

def clear(tail):
    self.head.next = self.tail
    self.tail.prev. self.head

garbage collection takes care of everything

Reading Notes 10.14.24

9.1. Binary Trees

Linked list: each node had < 1 successor

binary tree: each node has < 2 successors

- left child and right child

Definitions:

- Leaf: tree node w/ no children
- Internal node: node w/ <1 child
- Parent: node with a child
- Ancestors: node's parent, parent's parent, ...
- Root: the only node w/o a parent
  - the top node

Structure:

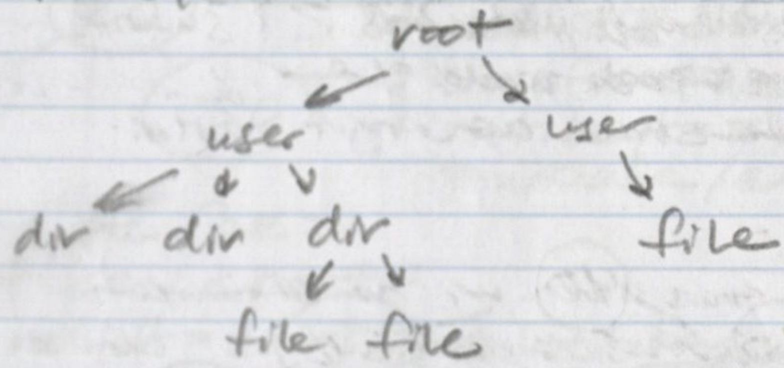
- edge: link from node to child
- depth: # of edges from root to node
- level: all nodes of the same depth
- height: largest depth of any node

Types of Binary Trees:

- full: every node has 0 or 2 children
- complete: if all levels (but the last) contain all possible nodes and all nodes in the last level are as far left as possible
- perfect: all internal nodes have 0 or 2 children and all leaf nodes are the same level

9.2. Applications of Trees

- file systems



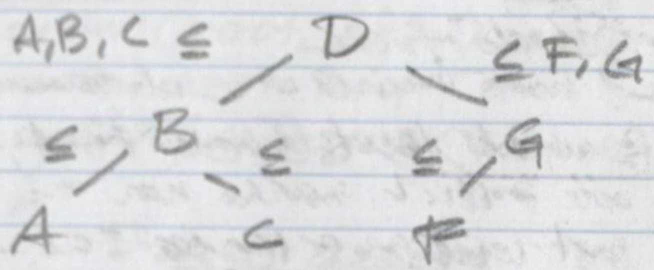
- Binary Space Partitioning (BSP)

  * useful for graphics rendering

- Collections of numbers

9.3. Binary Search Trees (BST)

- type of binary tree
  - any node's left subtree keys $\leq$ node's key
  - any node's right subtree keys $\geq$ node's key



- Searching: faster than a list

  worst case: $H + 1 \Rightarrow O(H)$

  H: height of tree

$$H = \lfloor \log_2 N \rfloor \Rightarrow O(\log N)$$

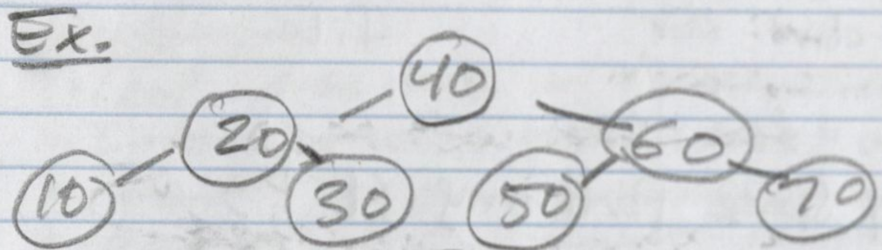
- successor: node that comes next in BST ordering
- predecessor: node that comes after in BST ordering

Lecture Notes 10.16.24

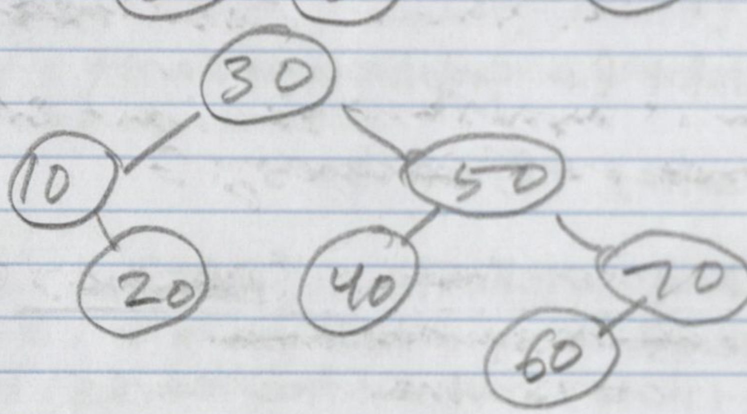
Binary Search Trees (BSTs):

- Assumes no duplicates
- Set is a common usage

Ex.

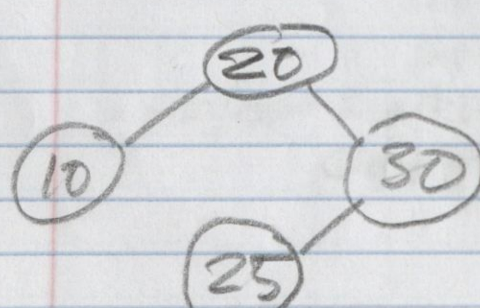


Ex.



- not full or complete
- binary tree
- binary search tree

BST Search (Recursive): search(root, key)



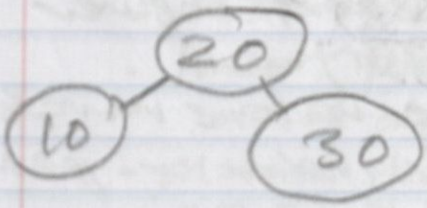
Base Case:

- if current Node is None or a match return current

Recursive Case:

if key < current
return search(current.left, key)
else return search(current.right, key)

BST Insert (Recursive): root = insert(root, key)



Base Case:
if root == None:
Node(key)
if root.key == key:
#update/exception/ignore

Recursive Case:
if key < root:
root.left = insert(root.left, key)

* examples of this in ZyBooks*

* order is important for how efficient a tree is (balancing)

BST Inorder Traversal (Recursive): inorder(root)

Base Case: empty tree
if root is None
return

Recursive Case:
inorder(root.left)
print root.key
inorder(root.right)

Lecture Notes: 10.18.24

Midterm Exam: 30 MCQs / short answer

- Basic VS-Code / Debugging
- What regions of memory are various variables in
- Move through an array by index or pointer
- ** pointers – what does it do?
- Dynamic arrays, Stacks, Sets, —
  - How to build stacks/sets on top of DA
- capacity vs. size?
- what is capacity/size after n insertions

* NONE on scanf or strtok *

- Debugging wrong code
- How did we implement our HW problems
  — why?
- Linear Search vs Binary Search
  — pros/cons?
- O(N) notation – what makes something $O(N^2)$?
  – what is a CTO – O(1)?
  – code snippet – what is complexity?
- What are the time and space complexity of the major sorting algorithms?
  – use of stack uses space
  – be able to identify sorting algorithms based on english descriptions
  – which work best with arrays? / LL? / DLL?

* • write code for insert before/after, removing, or traversing a LL or DLL
  — no half credit
  – know where it starts and ends

- pros/cons of dummy nodes
- when to use a LL vs. dynamic array vs. hash table

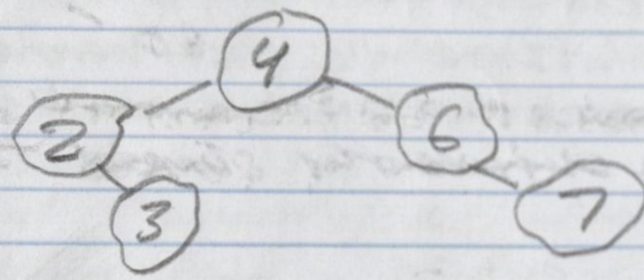
- know what a hash function is
- what is a collision ?
- linear probing vs. chaining
  - what do they do
- what are the datatypes of the hash struct ?
- ~~??~~ Bucket ?
- middle bits are the most random
- how does a hash table work ?

Lecture Notes: 10.28.24

Binary Search Trees in Python:

will be done mostly recursively

BST Search (Recursive): NO DUPS

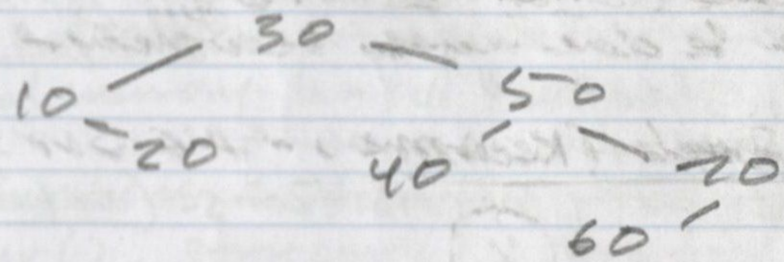


1. Base-Case: if current node is None
   return false
2. if key == current value
   return true
3. Recursive Case: if key < root value
   return search left
4. return search right

BST Insert (Recursive):

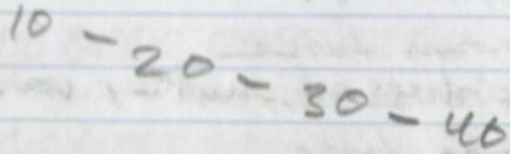
1. if root == None # no root
   return Node(key) # returns root
2. if key == root.key
   return root # have to return something
3. if key < root.key
   root.left = insert(root.left, key)
4. else root.right = insert(root.right, key)
   return root

Ex. 30, 50, 40, 10, 70, 20, 60



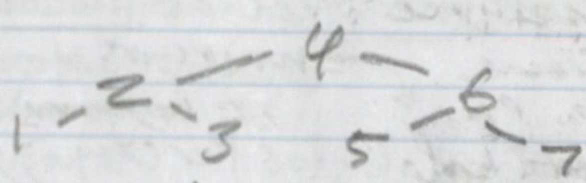
* these nodes in a different order would create a differently shaped tree

Ex. 10, 20, 30, 40



* lose log N search and insert with unbalanced trees

BST Inorder Traversal (Recursive):



* think of tree w/ 1 node then tree w/ 3 nodes

```
if root is None
   return
else
   inorder (root.left)
   print root value
   inorder (root.right)
```

Midterm Exam Review: 10.28.24

VS-code-terminal and debugging process

C file structure
→ multi-file programs
→ automating compilations (makefiles)

Assert Statements: assert(condition)
- exits program w/ error code and prints a message w/ line # if condition is false
- used in unit-tests

Pointers:
- variable holding address of another variable

reference operator: &
dereference operator: *
null pointer: points to nothing
  → cannot be dereferenced
void pointer: universal pointer
  → typically typecasted as needed

multiplication!

Remember:
```
malloc(numObj * bytesPerObj);
calloc(numObj, bytesPerObj);
realloc(void *p, numObj * bytesPerObj);
free(void *p);
```
dereference operator

can be used to create dynamically allocated arrays!

Memory Leak: lose access to previously allocated mem

Garbage Collection: automatically free() pointers to avoid memory-leaks (not C)

String Functions: strcmp(), strcpy(), strchr(), strrchr(), strstr()

Memory Structure:

Address:	Byte of data:	Var Name:
0xffff ffff ffff ffff	'A'	char c
...		
0x0000 0000 0000 0000	0x41	

* Address stores 64-bits
* 0x.... indicates hexadecimal digit (base 16)
* one hexadecimal digit holds 4 bits
* 1 byte = 8 bits = 2^8 different values

Data-type:	Size: bytes
bool	1
char	1
short	2
int	4
float	4
double	8
char *	8
int *	8

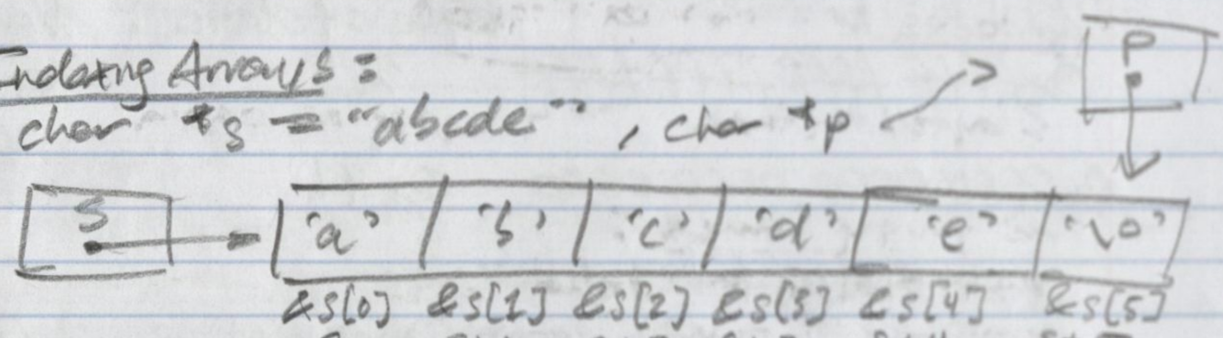
technically a bool would only require 1 bit of data but modern computer architectures are optimized to handle data in sizes of 1 byte or larger

Bit: smallest form of data possible
either a 0 or 1 (2 options)

Bit shifting:
Multiply by 2^n: value << n
Divide by 2^n: value >> n

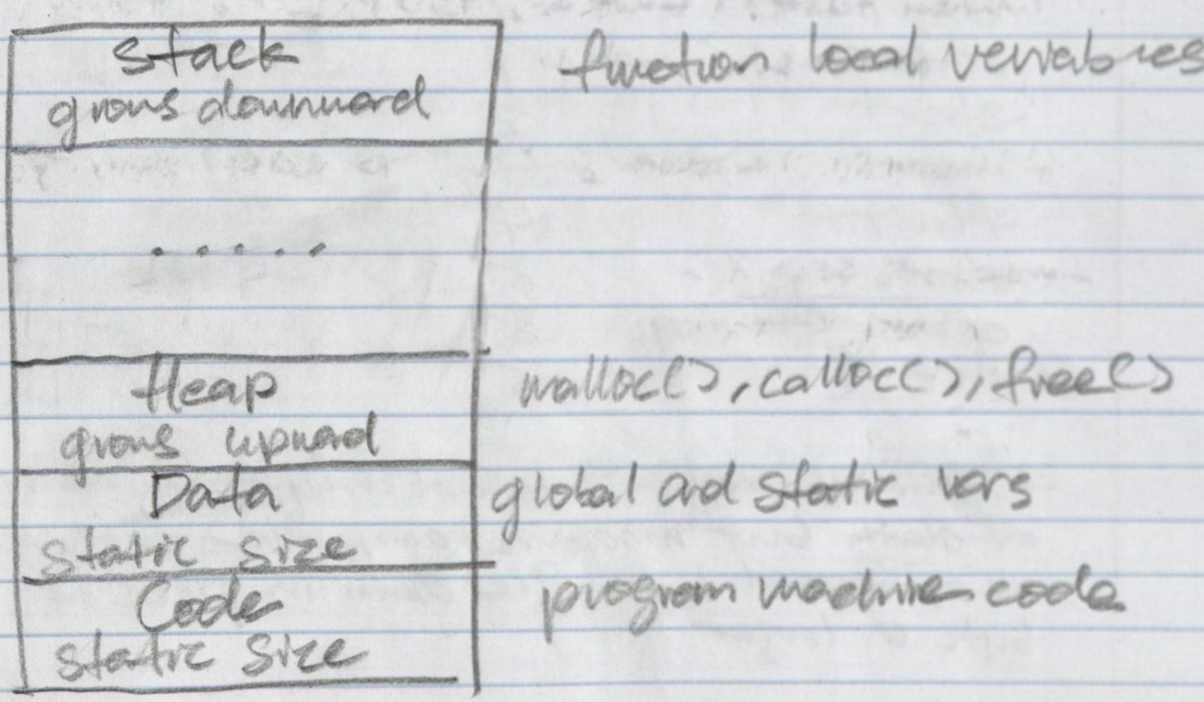
shifts all bits in a binary representation of a number by 1

Indexing Arrays:
char *s = "abcde", char *p



*p - s is the length of the array

Memory Management:



Array of char vs. String Constants:

```
char a[] = "cat";
char *p = "dog";
```

a: in stack, <u>can</u> change values but <u>not</u> reassign
'c' 'a' 't' '\0' - 4 bytes in stack

&p: in stack, "dog" in data, <u>can</u> reassign but <u>not</u> change values

p → $\underbrace{\text{'d' 'o' 'g' '\textbackslash 0'}}_{}$ → 4 bytes in data

8 bytes in stack

I/O Printing Strings:

```
char *string = "hello";
printf("%s\n", string);
puts(string); //does the same thing
```

I/O Reading Line-By-Line:

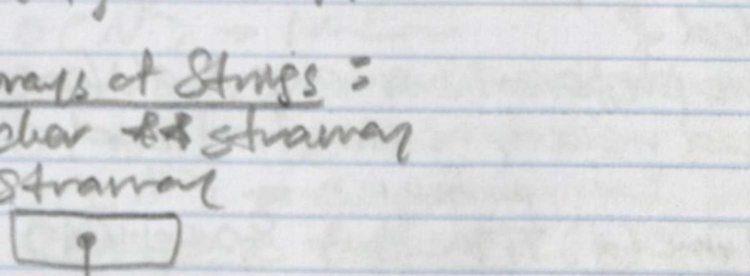
```
char buffer[BUFSIZ];
while(fgets(buffer, BUFSIZ, stdin)) {
    process(buffer)
}
```

* fgets() appends '\n' to each string *

Arrays of Strings:

```
char **strarray
```

strarray



Abstract Data Types : data type whose properties are specified independent of implementation

Stack : LIFO
Queue : FIFO
Deque : both LIFO and FIFO
Set : unique/unordered groupings

Data Structures : physical data structures used to build ADTs

dynamic array
linked list
hash tables

Dynamic Array :

```
typedef struct {
    int *data;       // internal array
    int capacity;    // total # elements
    int size;        // total valid elements
} Array;
```

* double the capacity and realloc(data, 2 * size) when needed *
→ realloc preserves memory and leaves unused memory uninitialized

Function	Time(A)	Time(W)	Space(A)	Space(W)
Append	O(1)	O(N)	O(1)	O(N)
At	O(1)	O(1)	O(1)	O(1)
Index	O(N)	O(N)	O(1)	O(1)
Insert	O(N)	O(N)	O(1)	O(N)

Stack Functions:

	Time	Space
push()	O(1)	O(1)
pop()	O(1)	O(1)
top()	O(1)	O(1)
empty()	O(1)	O(1)

Queue Functions:

	Time (A)	Space (A)
push()	O(1)	O(1)
pop()	O(N)	O(1)
front()	O(1)	O(1)
empty()	O(1)	O(1)

Set Functions:

	Time (A)	Space (A)
add()	O(N)	O(1)
contains()	O(N)	O(1)
remove()	O(N)	O(1)

Big O Notation:

$O(1)$ → constant time operation (CTO)
$O(\log N)$ → logarithmic
$O(N)$ → linear
$O(N \log N)$ → linearithmic
$O(N^2)$ → quadratic
$O(c^N)$ → exponential
$O(N!)$ → factorial

Recursive Algorithms:

algorithm that breaks a problem into subproblems and applies itself to those subproblems

Uses Recursive Functions:

a function that calls itself

Base Case: a case where the recursion stops

Recursive Case: a case where the function calls itself

Recurrence Relations:

used to calculate the time complexity of recursive algorithms

$$T(N) = xN + T(N/2)$$

refers to itself

*visualized w/ recursion graphs

Searching Algorithms:

Linear Search: searches from beginning to end of list for a key

Time: $O(N)$

Binary Search: searches ordered lists by recursively comparing a key to the middle value of a list

Time: $O(\log N)$ divide and conquer

Sorting Algorithms :

O(N^2) Algorithms : all take multiple linear passes through an array writing elements from an unsorted to a sorted region

Selection Sort : selects the smallest element from the unsorted region and puts it at the end of the sorted region

```
for (i=0; i<size-1; i++) {
  smallest = i;
  for (j=i+1; j<size; j++) {
     if (array[j] < array[smallest]) {
         smallest = j;
     }
  }

  temp = array[i];
  array[i] = array[smallest];
  array[smallest] = temp;
}
```

Insertion Sort: repeatedly inserts the next value from the unsorted region into the sorted region

```
for (i=1; i<size; i++) {
    j=i;
    while (j>0 && array[j] < array[j-1]) {
        temp = numbers[j];
        numbers[j] = numbers[j-1];
        numbers[j-1] = temp;
        j--;
    }
}
```

Bubble Sort: iterates through a list and swaps adjacent elements if the second element is less than the first element

```
for (i=0; i<size-1; i++) {
    for (j=0; j<size-i-1; j++) {
        if (array[j] > array[j+1]) {
            temp = numbers[j];
            numbers[j] = numbers[j+1];
            numbers[j+1] = temp;
        }
    }
}
```

* All have time: $O(N^2)$ *

* All have space: $O(1)$ *

Adaptive = algorithms that take advantage of data that is already partially sorted

- bubble : yes
- insertion : yes
- selection : no

Stable : does not change order of elements that are already in sorted order

- bubble : yes
- insertion : yes
- selection : no

Speed Comparisons =

1. insertion improved
2. selection
3. insertion
4. bubble
5. bubble improved

<u>Merge Sort</u>: divides a list into 2 halves, recursively sorts each half, and then merges sorted halfs

```
MergeSort(beg, end) {
  if (beg < end ) {
    mid = (beg+end) / 2;
    MergeSort(beg, mid);
    MergeSort (mid+1, end);
    for (i=beg; i<mid; i++) {
      temp[i] = array[i];
    }
    for (i=mid+1; i<end; i++) {
      j = end+mid+1-i;
      temp [j] = array[i];
    }
    i = beg;
    j = end;
    for (k = beg; k<end; k++) {
      if (temp[i]<temp[j]) {
        array[k] = temp[i];
        i++;
      }
      else {
        array[k] = temp [j];
        j--;
      }
    }
  }
}
```

Time: O(NlogN)
Space: O(N)
Adaptive: No
Stable: Yes

<u>Quicksort</u>: recursively partitions array into low and high parts (both unsorted) and recursively sorts those parts

```
QuickSort (beg, end) {
  i = beg;
  j = end;
  pivot = array [(i+j) / 2 ];
  while (i <= j) {
    while (a [i] < pivot) {
        i++;
    }
    while (a[j] > pivot) {
        j--;
    }
    if (i <= j) {
        swap (a[i], a[j]);
        i++;
        j--;
  }
  if (beg < j) {
    QuickSort (beg, j);
  }
  if (i < end) {
    QuickSort (i, end);
  }
```

	<u>Time</u>	<u>Space</u>
Best	O(NlogN)	O(logN)
Average	O(NlogN)	O(logN)
Worst	O(N²)	O(N)

Adaptive: No
Stable: No

List ADT: holds ordered data

Functions: append, prepend, insertafter, remove, search, print, printreverse, sort, isempty, getlength

Node:

```
typedef struct Node Node;
struct Node {
    int value;
    Node *next;
};
```

Linked-List: data structure of linked Nodes
* if link is lost data is lost forever *
  -> no name, no data

Methods:

```
Node *node_create(int value, Node *next) {
    Node *n = malloc(sizeof(Node));
    n->value = value;
    n->next = next;
    return n;
}

void node_delete(Node *n) {
    free(n);
}
```

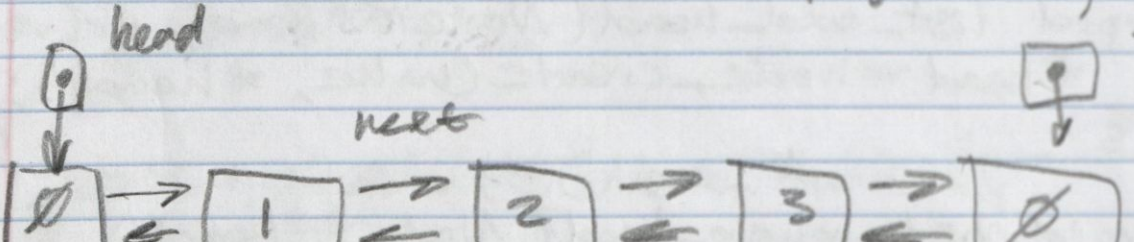
```
void list_print_iterative(Node *head) {
  for (Node *n = head; n != NULL; n = n->next) {
    printf("%d ", n->value);
  }
}

void list_delete_iterative(Node *head) {
  while (head != 0) {
    Node *n = head;
    head = head->next;
    node_delete(n);
  }
}

void list_add_after(Node *curr, int value) {
  curr->next = node_create(value, curr->next);
}

void list_remove_after(Node *curr) {
  Node *successor = curr->next->next;
  node_delete(curr->next);
  curr->next = successor;
}

void list_add_head(Node **head, int value) {
  *head = node_create(value, *head);
}

void list_remove_head(Node **head) {
  Node *n = *head;
  *head = (*head)->next;
  node_delete(n);
}
```

```
void list_print_recursive(Node *head) {
  if(head==NULL) return;
  printf("%d ", head->value);
  list_print_recursive(head->next);
}

void list_print_reversed_recursive(Node *head, Node *curr) {
  if(curr==NULL) return;
  list_print_reversed_recursive(head, curr->next);
  printf("%d ", curr->value);
}

void list_delete_recursive(Node *head) {
  if(head==NULL) return;
  list_delete_recursive(head->next);
  node_delete(head);
}
```

<u>Doubly Linked List</u> : also have a pointer to the previous node

* must keep track of this in methods as well

<u>Dummy Node</u> : nodes with NULL data at the head or tail to help with programming



* eliminates special cases — can always insert between 2 nodes *

Hash Tables : data structure that holds unordered items by mapping (hashing) each item to a location in an array

Key : value used to map to an array index
Bucket : hash table array element
Hash Function : computes bucket index from item's key
Collision : when 2 keys are mapped to the same bucket
Chaining : handles collisions by using a list for each bucket with multiple keys

Linear Probing : handles a collision by linearly searching subsequent buckets until an empty bucket is found

* has O(1) searching *

Load Factor : $\frac{\#\text{ items in HT}}{\#\text{ buckets}}$

Resize thresholds :
- Load factor
- open addressing (# collisions during an insert)
- chaining (size of bucket's LL)

Resizing :
- double the capacity
- allocate new array
- copy over values (re-insert)

Hash Table Struct:

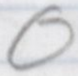
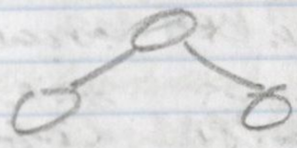
```c
typedef struct {
    Pair **buckets;
    size_t capacity;
    size_t size;
    double alpha;
} Table;
```

Pair Struct:

```c
typedef struct {
    char *key;
    long value;
} Pair;
```

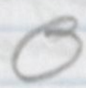
Lecture Notes: 11.1.24

Finding the number of nodes = len()

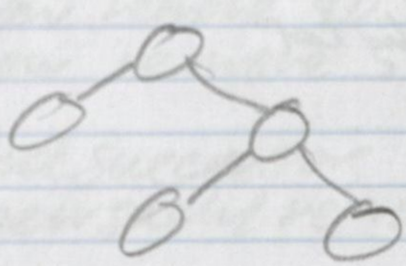
NONE  

```
if not root:
    return 0
left_len = len(root.left)
right_len = len(root.right)

return 1 + left_len + right_len
```

Find the height of a tree:

 NULL

height = 0 height = -1

 height = 2

```
if root == None:
    return -1
left_height = height(root.left)
right_height = height(root.right)

return max(left_height, right_height) + 1
```

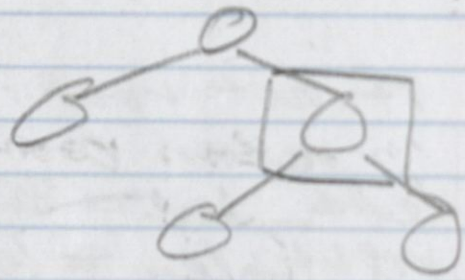
Removing a Node:

Base case:

    if Node == None
        return

Recursive Case:

find Node to remove



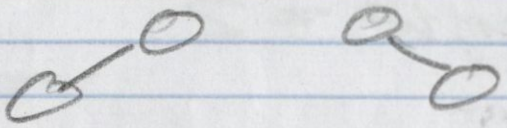
    if key < root.key
        root.left = remove(key, left)
    else if key > root.key
        root.right = remove(key, right

Case 1: leaf node

garbage collection takes care of free()

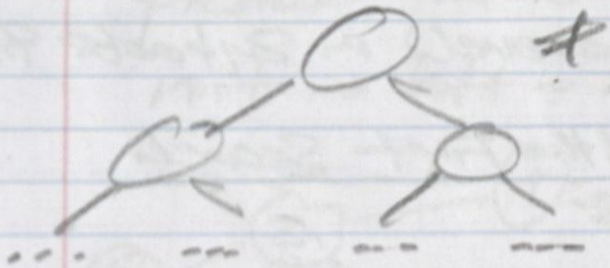
Just remove it: root = None

Case 2: root has 1 child



root = child

Case 3: root has 2 children

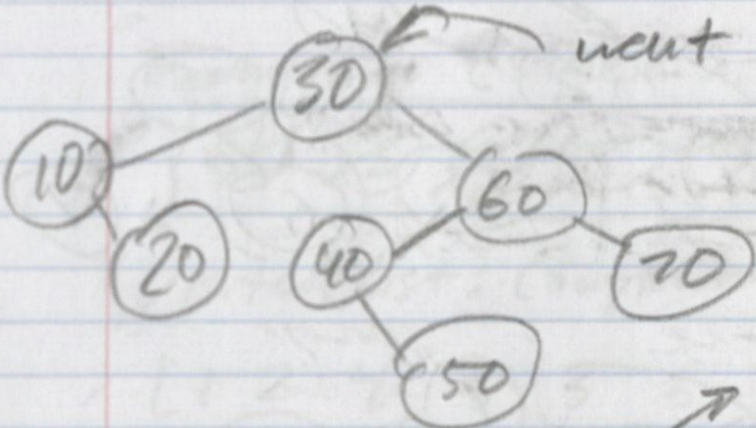


* replace root with successor *

→ arbitrary convention

Case 3+: root has 2 children, height > 1

want to remove



copy data over

remove 30 → replace with 40
now remove 40 → replace with 50
now remove 50

find successor
recursively remove it

* All examples in BST code ex in zybooks 9.11 *

Lecture Notes: 11.4.24

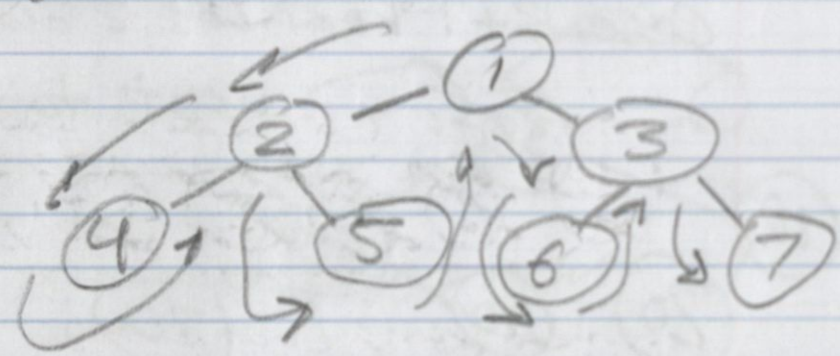
- * finished up node removal(s) in 2 parts 9/11 *

Depth-First and Breadth-First Search of Binary Trees:

- * not Binary Search Trees

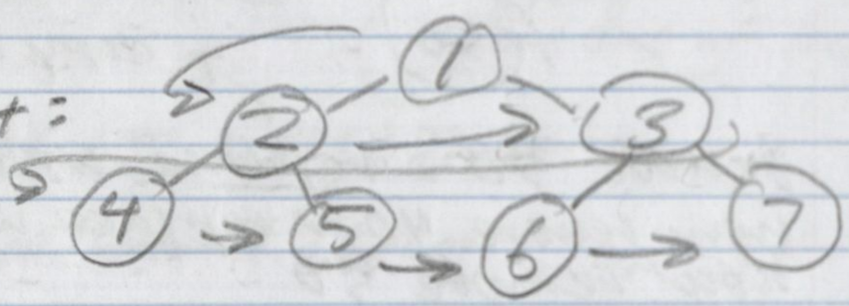
♡

Depth-first:



Breadth-first:

"level oriented":

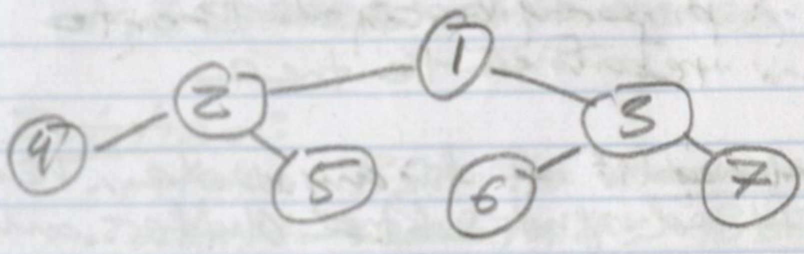


* if tree not complete put None in search

Iterative Depth-First Search:

preorder traversal: 1, 2, 4, 5, 3, 6, 7

root → left → right #recursive order



push
pop

Frontier: (stack) LIFO

~~1~~	~~3~~	~~2~~	~~5~~	~~4~~	~~7~~	~~6~~

Visited List: (output)

[1 2 4 5 3 6 7]

Bredth-First Search: 1, 2, 3, 4, 5, 6, 7

Frontier: (queue) FIFO

pop

~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~6~~	~~7~~

push

Visited List: (output)

left → right → root

[1 2 3 4 5 6 7]

* had to write recursively since its not Stack-oriented *

* BFS / DFS representations of graphs *

* Ex of this in Zybooks 9.12 *

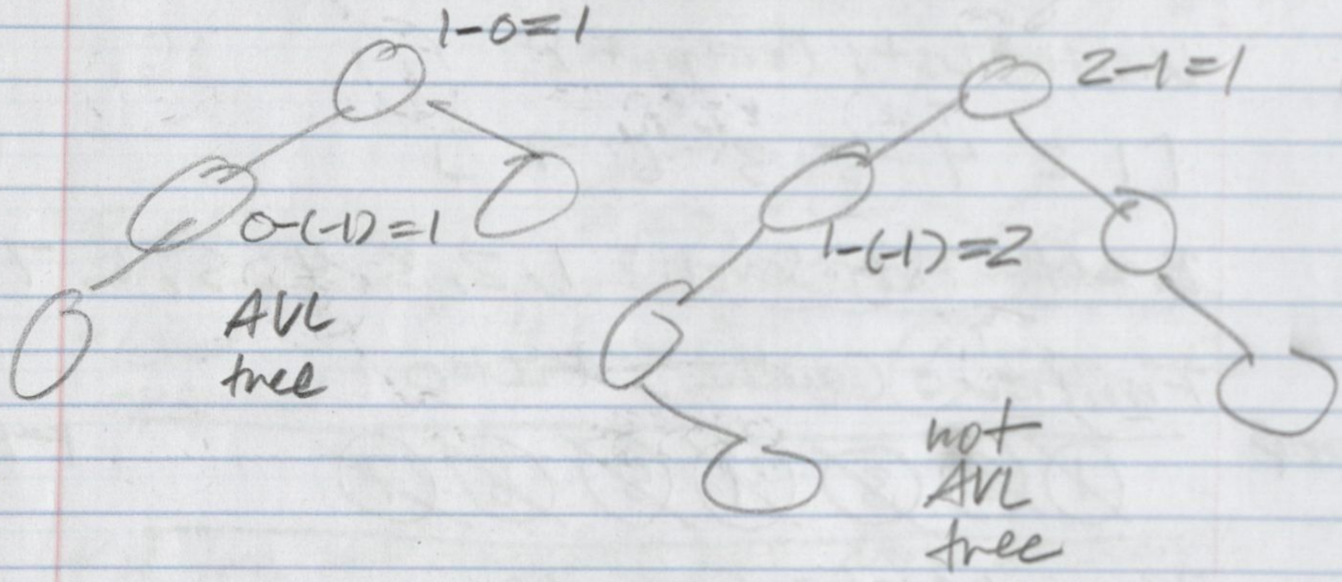
Reading Notes: 11.5.24

10.1. AVL: A balanced tree

AVL tree: BST tree with a height balance property and operations to rebalance the tree

height balanced: if for any node, height of left and right subtree differs only by 0 or 1

balance factor: left subtree height — right subtree height



recall: ○ height: −1

* an AVL tree does not always place nodes in perfect BST form with minimal height

→ but still $O(\log N)$ worst case

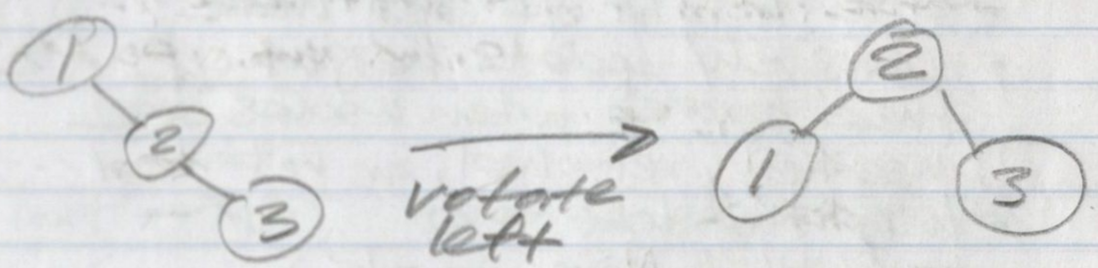
* no worse than 1.5 × minimum height

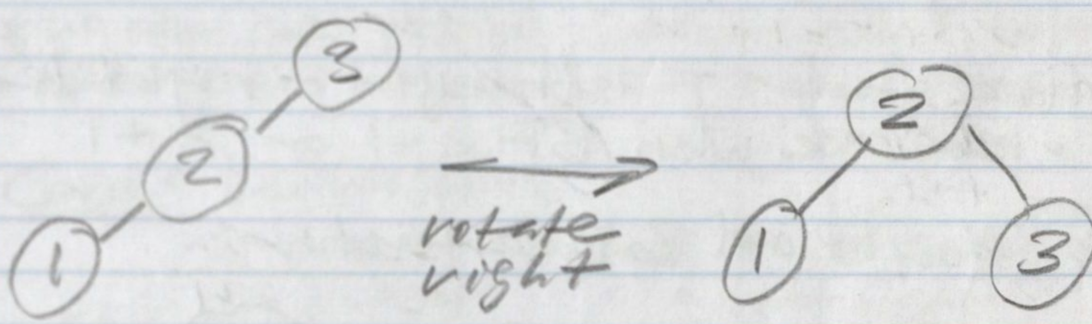
Lecture Notes: 11.6.24

AVL Trees: self balancing trees
- regular BST insert and remove
- rebalance after each

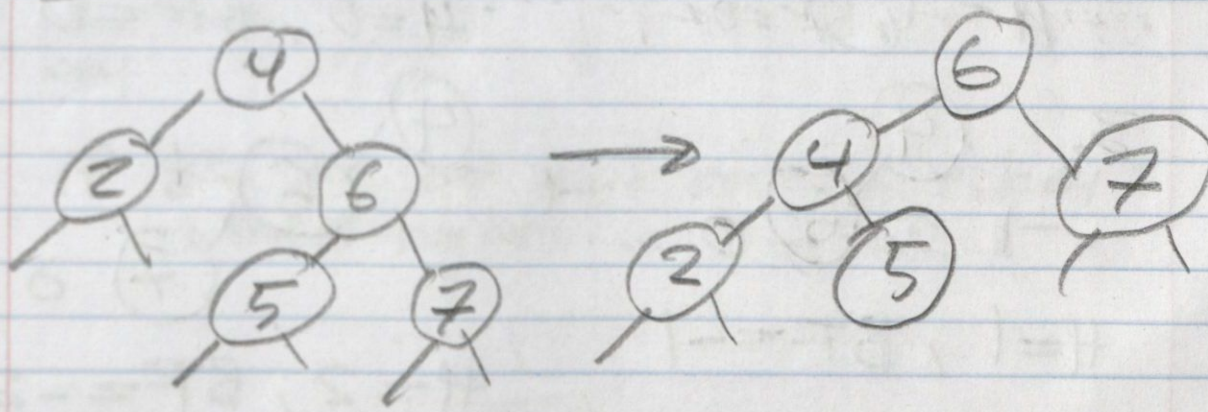
Rotations:
- rotate BST to maintain balance
- which maintains BST ordering properties

Simple Cases: rotate right and left

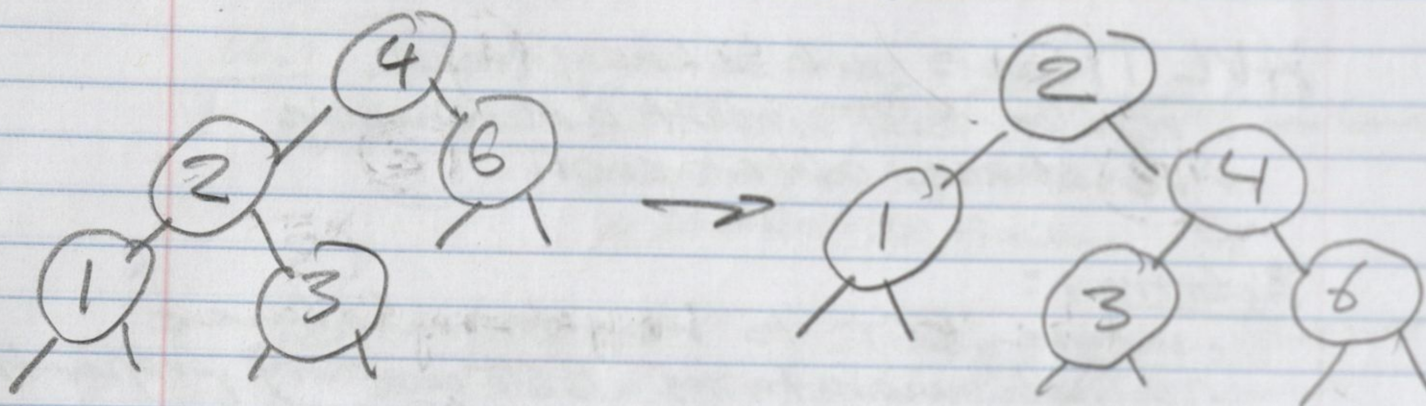




General Case: rotate left



General Case: rotate right



Detecting Imbalance:

- store height at each node
- recursively update height up the tree branch when nodes are inserted, removed, or rotated
- height of leaf $= 0$
- height of None $= -1$

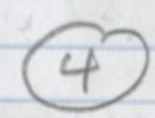
balance factor = left height − right height

- rebalance when $BF < -1$ or $> +1$

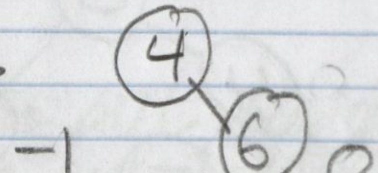
Ex. Height and Balance Factor

1. None

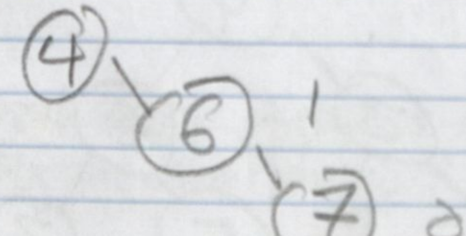
$H=-1, BF=0$

2. 

$H=0, BF=0$

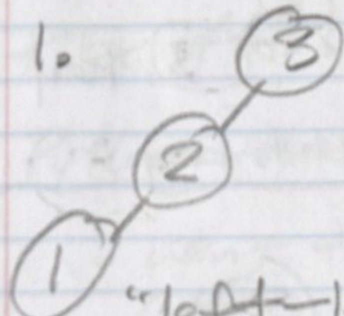
3. 

$H=1, BF=-1$

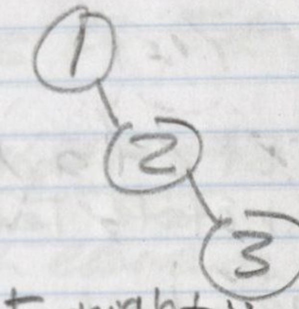


$H=2, BF=-2$

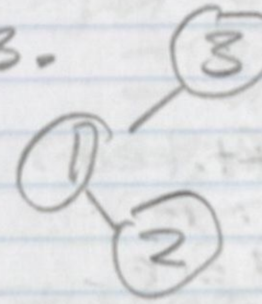
Imbalance: 4 Possible Cases

1. 

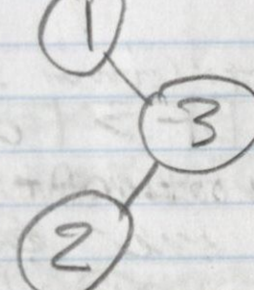
"left-left"

2. 

"right-right"

3. 

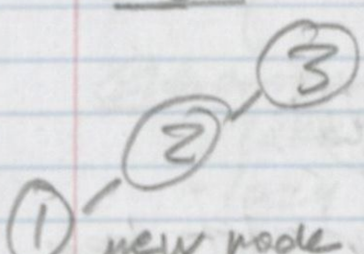
"left-right"

4. 

"right-left"

* no other cases possible because AVL's balance after every insertion, remove, or rotate

Case: left-left



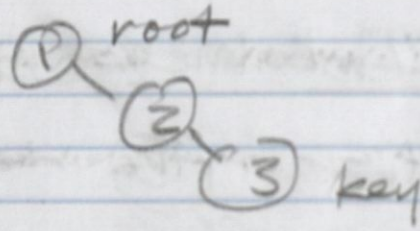
root pointer

new node key

how do we know it is a left left?

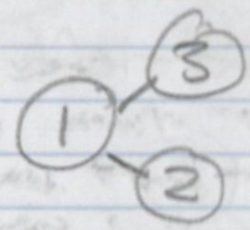
if BF > 1 and key < root.left.key : rotate right

Case: right right

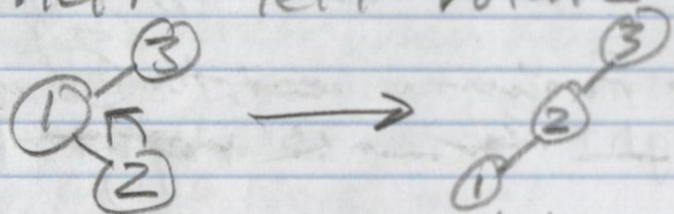


if BF < -1 and key > root.right.key :
    rotate left

Case : left-right

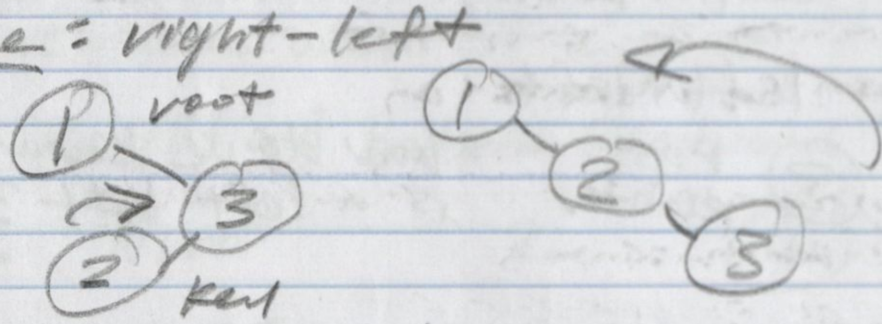


if BF > 1 and key > root.left.key :
    root.left = left rotate



    root = rotate right

Case : right-left



if BF < -1 and key < root.right.key
    root.right = rotate right
    root = rotate.left

Lecture Notes: 11.8.24

* Ex of AVL-tree in ZyBooks 10.5k

Python iterators, iterables, and generators:

- want to iterate through elements of (AVL) tree
- get them 1 at a time as we need them
  - not print all
  - not get full list
- want to be able to use "for ... in ..." protocol

Iterables: Python object that supports __iter__() method

Iterator: Python object that supports __iter__() and __next__() methods
- countable # of states
- can be iterated on

Generator: uses yield and next
- like functions
- lazy evaluation

Reading Notes: 11.10.24

11.1: Heaps

max-heap: complete binary tree where a node's key is $\geq$ its node's childrens keys

percolating: swapping a newly inserted node upwards until it does not violate the heap's ordering property

min-heap: same as a max heap but node's key $\leq$ node's children's keys

heaps are usually stored in arrays, but visualized in binary trees

Parent/Child indicing for a heap:

node index	parent index	child indices
0	N/A	1, 2
1	0	3, 4
2	0	5, 6
3	1	7, 8
⋮	⋮	⋮
i	$\lfloor (i-1)/2 \rfloor$	2*i+1, 2*i+2

Reading Notes: 11.12.24

11.3: Python: Heaps

- each level of a max-heap tree grows from left to right
  - a new level is added only when the previous level fills completely
- since tree is nearly filled completely, array implementation is optimal:

root: index 0
parent_index: (node_index - 1) // 2
left_child_index: 2 * node_index + 1
right_child_index: 2 * node_index + 2

// operator: integer division - divides and drops all decimals

exs of percolate_up() and percolate_down() methods

and insert() and remove() methods

insert():
- inserts at the end of the list
- percolates_up() to restore heap property

remove():
- returns root value
- percolate_down() to restore heap property

Reading Notes: 11.14.24

Heapsort: sorting algo that takes advantage of max-heap properties.
- repeatedly removes the max value
- builds sorted array in reverse order

heapify: converts array into heap
- leaf nodes satisfy max-heap property
- must percolate down every non-leaf node in reverse order

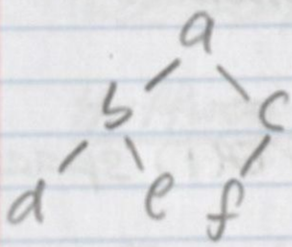
Largest Internal Node:

# nodes in binary heap	largest internal node index
1	-1 (no internal nodes)
2	0
3	0
4	1
5	1
⋮	⋮
N	$\lfloor N/2 \rfloor - 1$

more in section 11.6 on Priority Queues as well

11.15.24

Lecture Notes:
Array Representation of a Heap



[0]	[1]	[2]	[3]	[4]	[5]
a	b	c	d	e	f

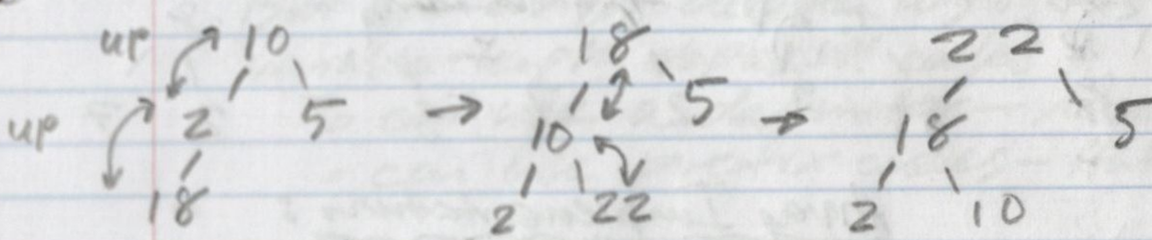
$left = 2 \cdot i_{parent} + 1$

$right = 2 \cdot i_{parent} + 2$

$parent = (i_{child} - 1) // 2$

MaxHeap: Push

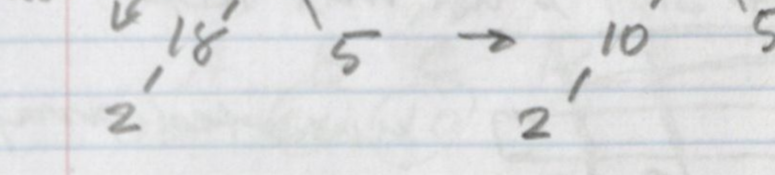
insert in order: 10, 2, 5, 18, 22



Array: [22, 18, 5, 2, 10]

MaxHeap: Pop

do one pop: 22



* ZyBooks 11.3 has good code examples *

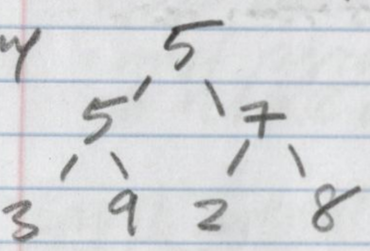
Heap Sort = O(NlogN)

1) push elements onto heap
2) pop them off 1 at a time
3) done

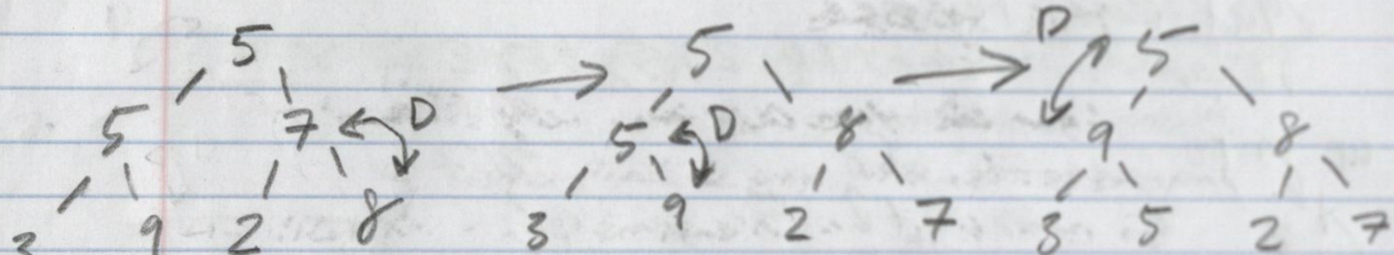
bonus: can be done in place ... O(1) space

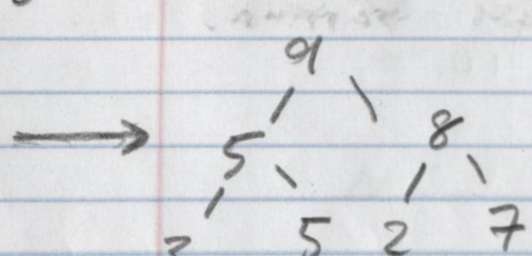
"Heapify-ing" an array:

given array:



percolate all non-leaf elements down to where they belong
- start at bottom
- index of parent of last element



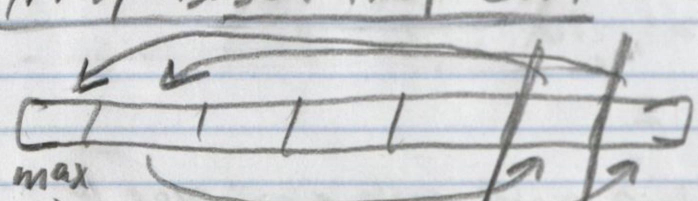


Array Implementation:

index of parent of last internal node down to 0

= size // 2 - 1 down to 0

Array-Based Heap Sort: use max heap



unsorted array

percolate down after every operation

* zybooks 11.5 has good ex of heap sort !!

→ sorts in ascending order

Lecture Notes: 11.18.24

#import heapq imports many functions to use heaps in Python

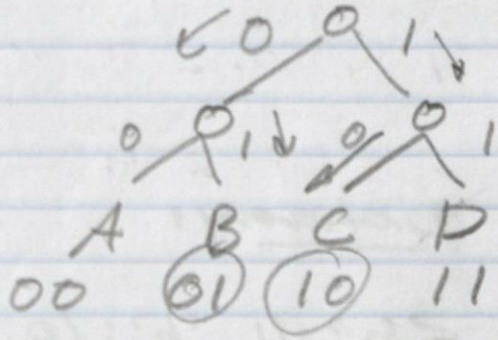
Huffman Codes: variable-length compression codes

Fixed Length Codes: ASCII, 24-bit RGB
- may be storing more bits than you need

Variable Length Codes:
- compresses data
- fast transmission
- less space

But how do you decode long streams of variable-length compressed codes?
- can use a delimeter — Morse Code
- can use prefix codes — Huffman Code

Prefix Codes: A, B, C, D
can use a binary tree

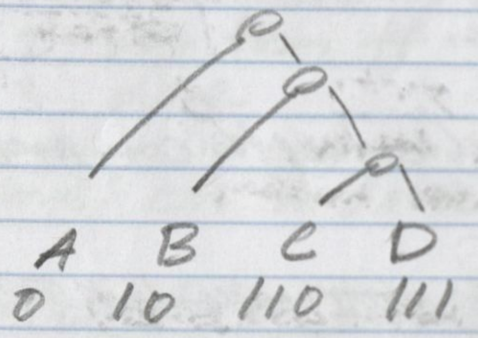


~~0110~~

BC

but this implementation is still fixed length

Prefix Codes: variable length



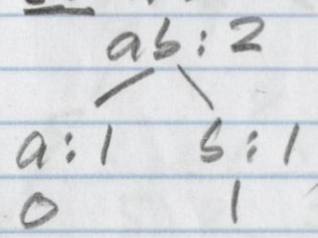
use an unbalanced binary tree

works well if letters have different frequencies of transmission

Huffman Codes: use letter frequencies to create an unbalanced tree that creates shortest path for most used letters

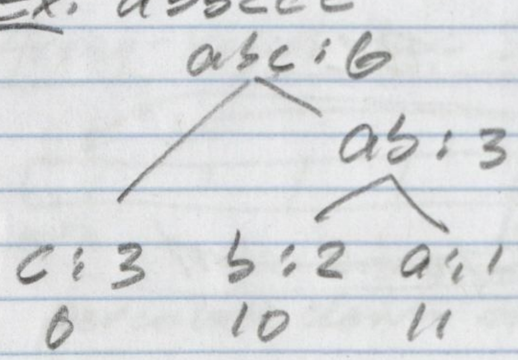
greedy algorithm → priority queue → heap

Ex. ab



Dictionary:
{'a': 0, 'b': 1}

Ex. abbccc



Dictionary:

{'a': 11, 'b': 10, 'c': 0}

Huffman Encoding Algorithm:

- Get Frequencies
- Build tree
  - Push nodes onto min priority queue by value
  - Pop 2 lowest value nodes (highest priority)
  - create new node with combined values and keys
  - push new node with leaves
  - repeat until 1 node

Node

$\boxed{\text{Symbol} \\ \text{Freq}}$

Ex. aabbcccddd

abcd:10

cd:6 ab:4

c:3 d:3 a:2 b:2

PQ:

~~a:2 b:2~~

~~c:3 d:3~~

ab:4

~~cd:6~~

abcd:10

Reading Notes: Intro to Graphs 11.19.24

graph: DS that represents connections among items

vertex: or node, item in a graph

edge: connection b/w two vertices

Adjacency and Paths:

- 2 vertices are adjacent if connected by an edge
- a path is a sequence of edges leading from a source vertex to a destination vertex
- path length: # edges in a path
- distance b/w 2 vertices is their shortest path length

Applications: geographical maps and navigation, product recommendations, social networks

Adjacency List: represents a graph w/ each vertex having a list of adjacent vertices, each list item representing an edge

size: $O(V+E)$

Sparse Graph: similar but w/ less than maximum edges

Adjacency Matrix: each vertex has a matrix row and column; a 1 denotes a connecting edge and a 0 means no connecting edge

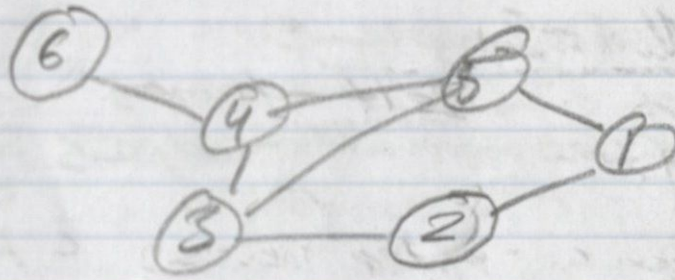
Space: $O(V^2)$; $O(1)$ access

Lecture Notes: 11.20.24

* directed acyclic graph → tree
trees are an application of graphs

Graph: set of Nodes (vertices) and edges

$G = (V, E)$



edge: source → target

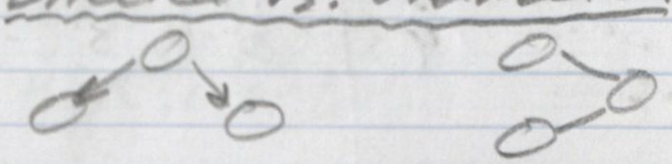
$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{\{1,2\}, \{2,3\}, \{3,4\}, \{4,5\}, \{3,5\}\}$

Applications:
- maps, GPS, pathfinding
- recommendations
- social networks
- city infrastructure
- problem solving: cracking a code
- neural networks

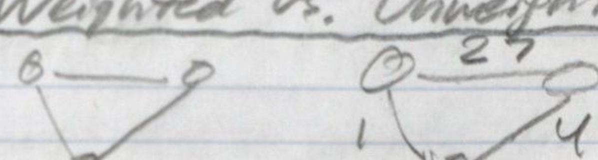
Properties:

Directed vs. Undirected

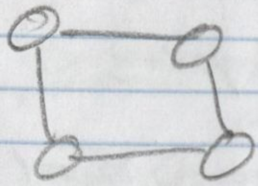


arrows imply directions

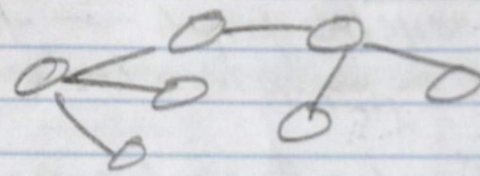
Weighted vs. Unweighted



Cyclic vs Acyclic:

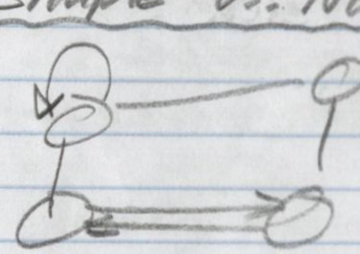


cyclic



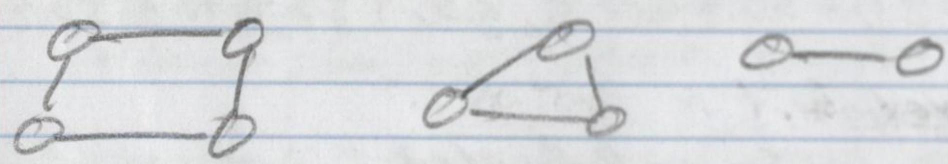
acyclic

Simple vs. Non-Simple:

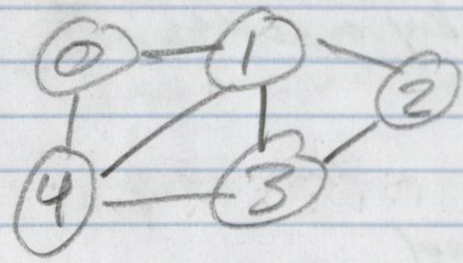


- self-loops
- multi-edges

Connected vs Disconnected:



Agacency List Representations:

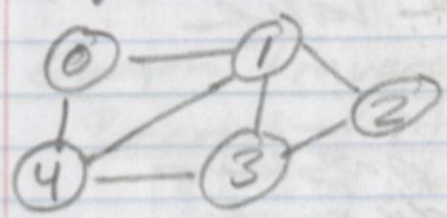


Vertex	Neighbors
0	1, 4
1	0, 2, 3, 4
2	1, 3
3	1, 2, 4
4	0, 1, 3

Data Structure:

```
{
    Vertex: [ ]
    :
}
```

Edge List File Format:



Ex.

5 7 num_node num_edges
0 1 source1 target1
1 2 source2 target2
2 3 source3 target3
1 3 :
1 4 :
3 4 :
4 0

Adjacency List in Python:

```
for each edge:
    graph[Source].append[target]
    graph[target].append[source]
```

both for undirected graphs

```
graph = {
    0: [1, 4],
    1: [0, 4, 3, 2],
    2: [1, 3],
    3: [4, 1, 2],
    4: [0, 1, 3]
}
```

* sort keys and lists for readability

* must have an empty list before appending

defaultdict = Python library that lets you define a default value datatype

solves the d["node"] problem if the node doesn't already exist

from collections import defaultdict

↳ has a lot of unique data structures

example of reading in a graph from an edge list into a default dict in ZyBooks 12.5

Reading Notes: 11.21.24

Graph Traversals:

Breadth-first search (BFS):

visits a starting vertex
then all vertices at distance 1
then distance 2
⋮
* built on a queue *
⋮
until all have been visited once

Application: Social network recommendations

when BFS first encounters a Node it is **discovered**
the vertices in **queue** are called the **frontier**

Depth-first search: visits a starting vertex, and every vertex in that path until its end, and then backtracks ...

* built on a stack *

Directed Graph: digraph
has directed edges (arrows)

Appl: airline flights, course registration

* nodes are only adjacent up-stream

$A \rightarrow B$

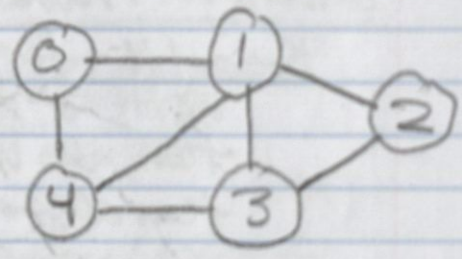
b adjacent to A ; A **not** adjacent to B

path, cycle

Lecture Notes: 11.22.24

Adjacency Matrix Representation:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	1	0	0



* matrix operations can be used in algorithms

Adjacency Matrix in Python:

* no 2D array type unless you import numpy ect...

```
graph = [
    [0,1,0,0,1],
    [1,0,1,1,0],
    [0,1,0,1,0],
    [0,1,1,0,1],
    [1,1,1,0,0]
]
```

use a list of lists:
- initialize matrix [v][v]

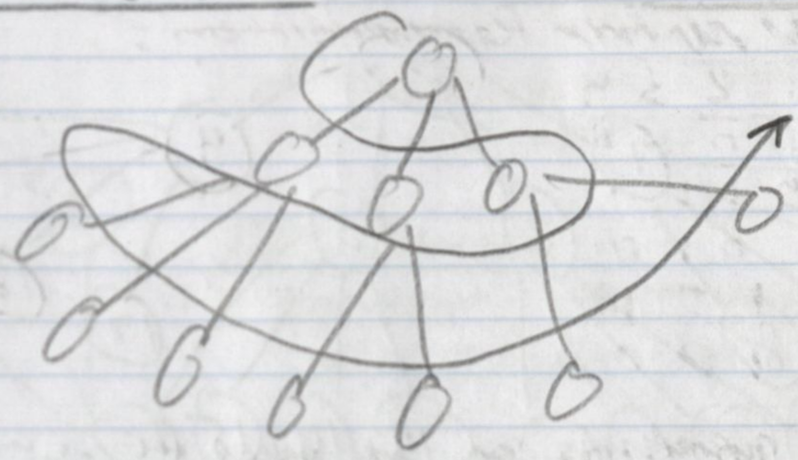
for each edge:
  matrix [source] [target] = 1
  matrix [target] [source] = 1

* can be read in in an edge list file format

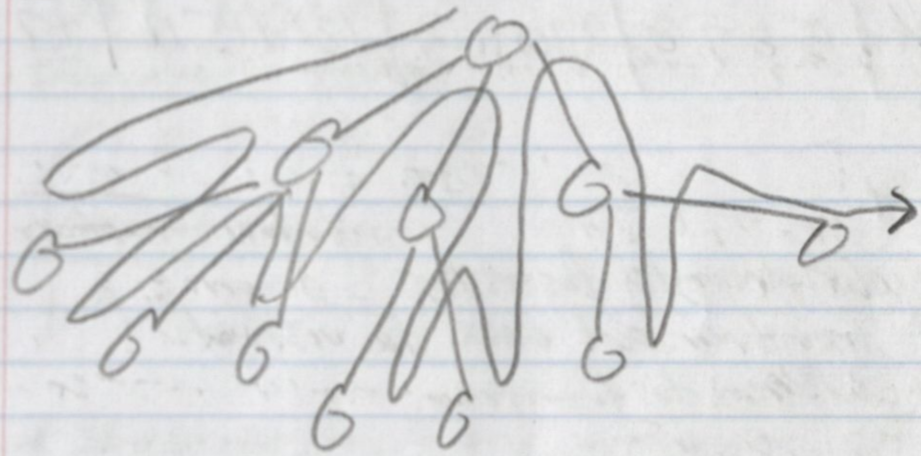
* example of this in ZyBooks 12.5

* just use matrix [source] [target] = 1 for directed graphs

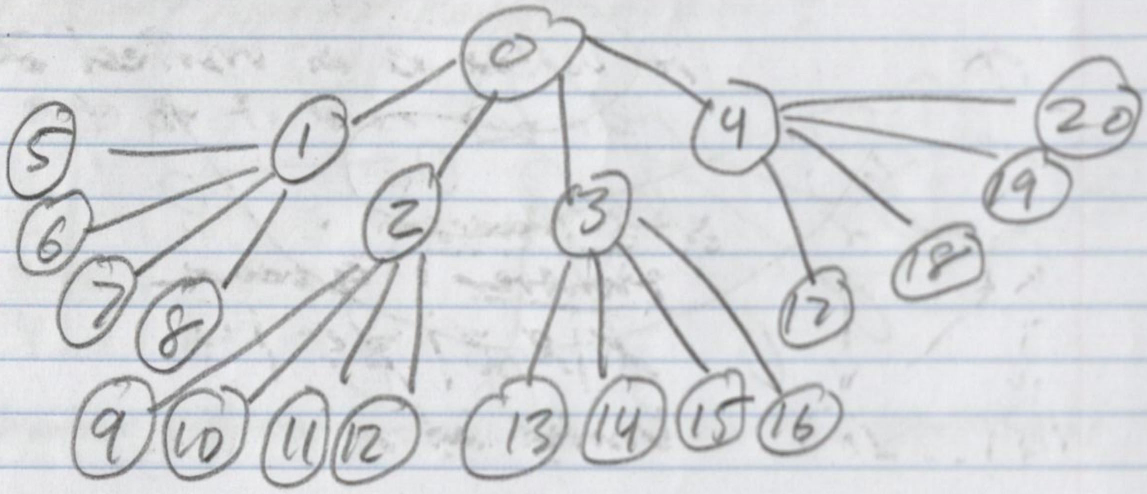
Bredth-first search:



Depth-first search:



Breadth-first search:



Frontier: queue (FIFO)

~~0~~ | ~~1~~ ~~2~~ ~~3~~ ~~4~~ | 5 6 7 8 | 9 10 11 12 | 13 14 15 16 | 17 18 19 20

Visited:

0 1 2 3 4 . . .

- add all children to frontier
- pop off frontier and add to visited
- add all children to frontier

Depth-first search:

Frontier: stack (LIFO)

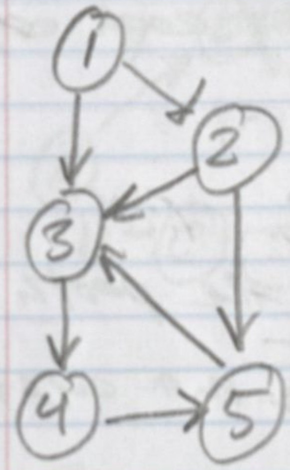
~~0~~ | 1 2 3 ~~4~~ | 17 18 19 ~~20~~ |

Visited:

0 4 20 . . .

same algorithm but with a stack

Dealing with Cycles: use a set



if vertex is in visited set:
do not add it to the frontier

BFS Traversal:

Frontier: queue

~~1~~ | ~~2~~ ~~3~~ | ~~3~~ 5 | 4

Visited Set:

1 2 3

- - -

DFS Traversal:

Frontier: stack

~~1~~ | 2 ~~3~~ | ~~4~~ | ~~5~~

Visited Set:

1 3 4 5

* Zybooks 12.9 has examples of these *
→ not directed

* uses the collection library *
→ has a deque class
- highly optimized
- circular buffer
- doubly-linked lists of static blocks of memory

Reading Notes: 11.24.24

12.10: Weighted Graphs

graph with a weight or cost at each edge

* may be directed or undirected

Path Length in Weighted Graphs:
sum of the edge weights in the path

Cycle length: sum of edge weights in a cycle

Negative Edge weight cycle:
cycle w/ a cycle length less than 0
shortest path does not exist

12.11: Dijkstra's shortest path

algorithm that determines the shortest path from a start vertex to each vertex in a graph

distance: shortest path from start vertex

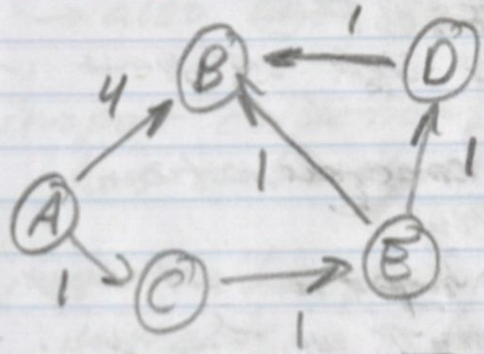
predecessor pointer: points to the previous vertex along the shortest path from the start vertex

* more detailed description of algorithm in textbook *

Runtime: $O(V^2)$
# of vertices in graph

* algorithm does not work w/ negative edge weights

Lecture Notes: 11-25-24



Adjacency List:

{
A: {B:4, C:1}
B: { }
C: {E:1}
D: {B:1}
E: {D:1, B:1}
}

dictionary of dictionaries

Review: DFS

* weights do not matter *

1. add first node to frontier
2. pop node and add all adjacent nodes to frontier
3. add popped nodes to visited list

* DFS → stack

* BFS → queue

frontier: what are we visiting next?

visited: what have we already seen?

Single-Source Shortest Path:

- BFS/DFS tell us if there is a path to any other vertex
- these tell us the distance

Dijkstra's Shortest Path:

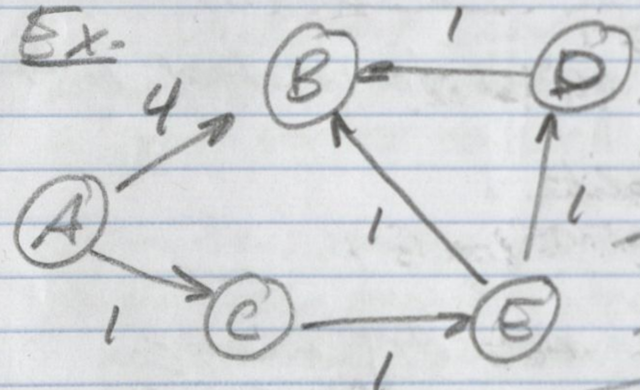
- very similar to DFS/BFS
  - uses frontier
  - uses visited
- takes weights into consideration
- uses greedy algorithm
  - take best looking option at any point

frontier: uses a priority queue by min-heap
contains (accumulated distance, vertex)

running sum

visited: dictionary
{ vertex : total distance }

Ex.



Frontier	Visited
~~(0, A)~~	—
~~(4, B)~~	A : 0
~~(1, C)~~	C : 1
~~(2, E)~~	E : 2
~~(3, B)~~	B : 3
~~(3, D)~~	D : 3

Pseudocode:

```
while frontier is not empty:
    remove (dist, vertex) from frontier
    w/ lowest distance
    if vertex not visited:
        store (dist, vertex) in visited
        for each non-visited neighbor
            add to frontier (total dist, neighbor)
```

* example of this in ZyBooks 12.12, Ex: 33
→ also has DFS / BFS

import collections
has a bunch of elementary data structures

import heapq
imports a heap, can build priority queue on top of this

Lecture Notes: 12.2.24

* comments on homework *
* Ex of Dijkstra's Algorithm *

Dijkstra's Alg Time Complexity
w/ Binary Heap:
Push / Pop: $O(\log N)$

Vertex + Edge:
$O(V) + O(E \log E)$

Revised Dijkstra's Alg:
* want to return actual paths, not just distances *
→ for homework
→ change visited and frontier

Visited Before: { Vertex : Distance }
After: { Vertex : Source }

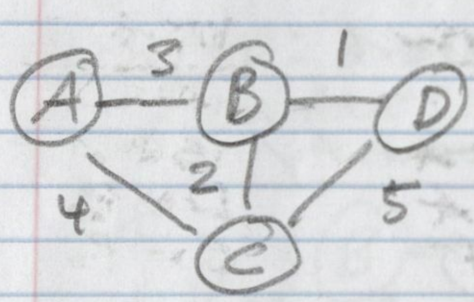
Frontier Before: { Distance : Vertex }
After: (Distance, Vertex, Source)

Minimum Spanning Tree (MST):

shortest possible tree that still connects all nodes

Prim's Algorithm: alg to create a MST, exactly like Dijkstra's alg, but picks the very shortest edge every time instead of starting from the base

Ex. Dijkstra's w/ path



Frontier (D, V, source)	Visited (V, source)
~~0 A←A~~	A←A
~~3 B←A~~	B←A
~~4 C←A~~	C←A
~~4 D←B~~	D←B
5 C←B	
9 D←C	

Reconstructing Path: work backwards

A : A → A
B : A → B
C : A → C
D : A → B → D

MST:
- undirected
- hits all nodes
- minimal length

Prim's Algorithm for MST:

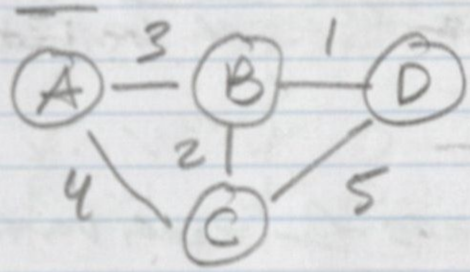
- start w/ Dijkstra's alg w/ edges
- choose next vertex as destination of shortest overall edge

Prim's Frontier and Visited:

Visited: Dictionary
{ vertex : Source }

Frontier: List
[(distance, vertex, source), ...]

Ex.



Frontier	Visited
~~0 A←A~~	A←A
~~3 B←A~~	B←A
4 C←A	D←B
~~1 D←B~~	C←B
~~2 C←B~~	
5 C←D	

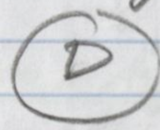
Prim MST vs Dijkstra SSSP:

- Dijkstra depends on origin
- Prim does not
- path traveleted will be different

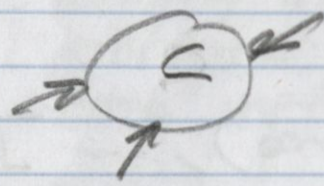
* examples of these in ZyBooks 12.15
Ex34 *

Lecture for 12/4: 12.9.24

Degree of a Vertex: number of edges pointing to a vertex



degree 0



degree 3

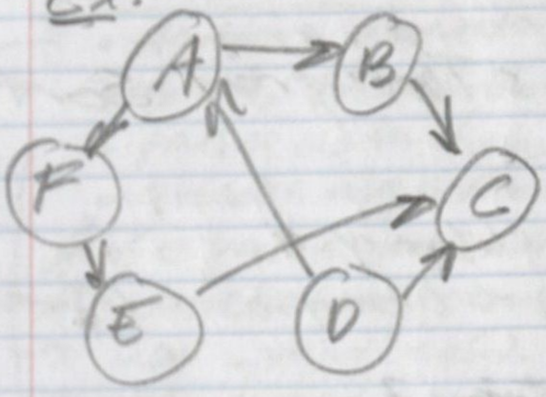
Kahn's Topological Sort Algorithm:

- table of the degree's of each vertex
- frontier : set

pseudocode:

```
create table of degrees for each vertex
while frontier not empty:
    pop vertex from frontier
    append to sorted
    for each neighbor of vertex:
        decrement degree
        if degree == 0:
            add to frontier
```

Ex.



Frontier	Visited
~~D~~	D
~~A~~	A
~~F~~	F
~~B~~	B
~~E~~	E
~~C~~	C

Degree

A	~~1~~ 0
B	~~1~~ 0
C	~~3~~ ~~2~~ ~~1~~ 0
D	0
E	~~1~~ 0
F	~~1~~ 0

Detecting cycles:

if # visited < # vertices in graph:
  there was a cycle

Complexity of Kahns:

$O(V+E)$

\# vertices ? ≈ # edges

* use of the default dict is common w/ Kahn's Alg *

* ex in ZyBooks 12.17 Ex 35 *

Lecture for 12/6: 12.9.24

Ex. Makefile w/ Kahn's Topological Sort

```
library.o: library.c library.h
	$(CC) $(CFLAGS) -o library.o -c library.c

main.o: main.c library.h
	$(CC) $(CFLAGS) -o main.o -c main.c

program: main.o library.o
	$(CC) $(LDFLAGS) -o program main.o library.o
```

Dependency Graph:

$$
\begin{array}{ccccc}
\text{library.c} & & \text{library.h} & & \text{main.c} \\
\searrow & \swarrow & & \searrow & \swarrow \\
& \text{library.o} & & \text{main.o} & \\
& \searrow & & \swarrow & \\
& & \text{program} & &
\end{array}
$$

Parsing:

```
for each line in Makefile
  if ":" not in line:
    Split line around ":" to src, target
    for each source:
      update target in adjacency list
```

*ex in Zybooks 12.17.1

Lecture Notes: Exam Review 12-11-24

- iterators, generators, list comprehensions
- yield, yield from
- parsing data from STDIN
- greedy algorithms - cups homework
- * c vars in memory - from midterm
- * dereferencing double pointers - from midterm
  - → use .readline(). split()
  - → use map() or list comprehensions
  - → __iter__(), __next__()
  - → for ... in ... :
- Python doubly-linked list
- Binary trees, binary search trees
  - → leaf node, root node, parent, child
  - → height = # of edges (empty tree = -1)
  - → full, complete, perfect trees
- Heaps - must be complete to represent as an array
- BST rules, search, insert, remove, inorder traversal, height/insertion order
  - → removal - find a successor
    - go to right child and find left-most leaf node - swap
- DFS/BFS; DFS: stack, BFS: queue
- AVL trees
  - → rotations, insertions, removals, balance factor
    - → must preserve BST properties
    - → left-left, left-right, right-right, right-left
  - → not always faster than BST
- Heap
  - is tree a heap? is tree a BST?
  - needs to be complete

- heaps stored as arrays
  - formula or derive from tree
- priority queue
- huffman compression
  - tree, letter frequency, greedy algorithm
  - least frequent first, priority queue

Summary of Graph Algorithms:

Algorithm	Frontier	Visited	Other?	Complexity
DFS	Stack	List		$O(V+E)$
BFS	Queue	List		$O(V+E)$
Dijkstra's SSSP	Priority Heap	Dict		$O(V+E\log E)$
Prim's MST	Priority Heap	Dict		$O(V+E\log E)$
Kahn's TS	Set	List	Degrees	$O(V+E)$

Final Exam Review: 12.13.24

Python: true OOP Language
- everything in Python is an object of a type w/ methods
    dir(type) # lists methods of a type

Regular methods: object.method()
Internal methods: object._method()
Magic methods: object.__method__()

Python Debugging and Unit Tests: import unittest

Doubly Linked List Class: DLL w/ dummy head and tail nodes and methods

Reading Data from STDIN:

```
def main(stream=sys.stdin):
    for line in stream:
        input_line = line.strip()          → removes '\n'
        input_list = [int(x) for x in input_line.split()]
        input_string = ' '.join(input_list)
```

→ delimits by ' ', parsing data into a list
→ joins list back into a string

Ex. clearing a Python DLL

```
def clear(tail):
    self.head.next = self.tail
    self.tail.prev = self.head
```

→ garbage collection takes care of everything

Binary Trees:

Vocab: leaf node, internal node, parent node, ancestor nodes, root node

Structure: edge, depth, level, height

Types: full, complete, perfect

Binary Search Trees:

- assumes no duplicate nodes
- commonly implemented recursively

Methods: search, insert, inorder, len, remove

Ex. BST Inorder Traversal

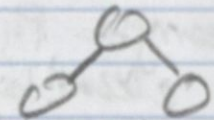
```
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.key)
    inorder(root.right)
```

Tree length: # of nodes

Tree height: "depth" of tree

NULL — height = -1

O — height = 0



height = 1

Node Removal:

- different for node w/ 0, 1, or 2 children
- with 2 children:
  - → recursively replace node with successor
    - leftmost leaf node of right child

Depth-First Traversal:
preorder traversal: root → left → right
inorder traversal: left → root → right
postorder traversal: left → right → root

Frontier: stack
Visited List: list

Bredth-First Search:
level-order traversal

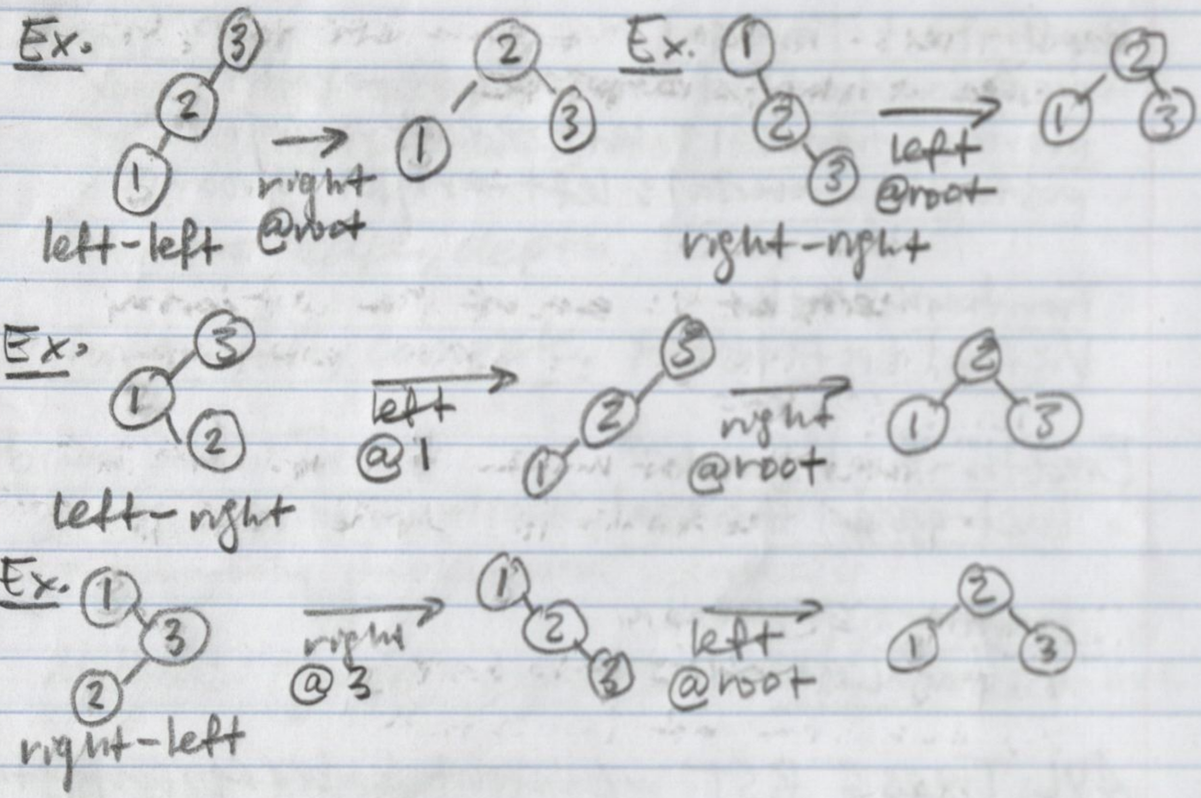
Frontier: queue
Visited List: list

AVL Trees: BST w/ height balancing properties and rebalancing methods

balance factor: left subtree height − right subtree height

Rotations: help maintain balance
4 Cases:
- left-left
- right-right
- left-right
- right-left

Detecting Imbalances:
- store BF at each node
- recursively update BFs up the tree branch when a node is inserted, removed, or rotated



Python Iterable: object that supports `__iter__()`

Python Iterator: object that supports `__iter__()` and `__next__()`

Python Generator: iterator that lazily evaluates, using yield and yield from and supporting `__next__()`
- yield - like return
- yield from - yield value from another generator

Heaps:

min/max-heap: complete binary tree where any node's key ≤/≥ all its node's children's keys

percolating: swapping a newly inserted node upwards until it does not violate its ordering property

* stored in arrays and visualized in binary trees *

node_index	parent_index	child_indices
i	$\lfloor (i-1)/2 \rfloor$	$2*i+1,\ 2*i+2$

* each level of a heap grows from left to right &
– tree is always complete

<u>methods</u>: insert(), remove()

<u>insert()</u>: push
- inserts at the end of the list/array
- percolates <u>up</u> to restore heap property

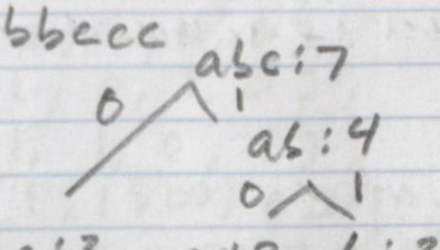
<u>remove()</u>: pop
- returns root node and swaps with last val in list
- percolates down to restore heap property

<u>Heap Sort</u>: $O(N \log N)$
1) push elements onto heap
2) pop them off 1 at a time
3) done

<u>Huffman Codes</u>: variable-length compression code
1) Get frequencies
2) Build a tree
- push nodes onto a min priority queue
- pop 2 minimum values
- push combined values and keys as new node
- repeat until 1 node contains all values

<u>Prefix Codes</u>:
- balanced trees – fixed length
- unbalanced trees – variable length

<u>Ex.</u> aabbccc



<u>Dictionary</u>:
{'a':11, 'b':10, 'c':0}

Graphs:

Vocab: vertex/node, edge, adjacent edges, path, path length, distance, degree

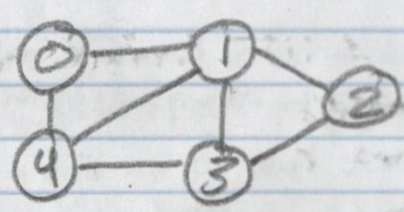
ex. $V = \{1, 2, 3, 4\}$

$E = \{\{1,2\}, \{2,3\}, \{3,4\}\}$

Properties:

- directed vs undirected
- weighted vs unweighted
- cyclic vs. acyclic
- simple vs nonsimple
  - self loops, multi-edges
- connected vs disconnected

Ex-



Edge List File Format

5	7	num_nodes	num_edges
0	1	source1	target1
1	2	source 2	target 2
2	3	⋮	⋮
1	3		
1	4		
3	4		
4	0		

Adjacency List Representation:

```
for each edge: # from ELF
    graph[source].append[target]
    graph[target].append[source]
```

both for undirected graphs

```
graph = {
    0: [1,4],
    1: [0, 4, 3, 2],
    2: [1, 3],
    3: [4, 1, 2],
    4: [0, 1, 3]
}
```

Size: $O(V+E)$

use another dict here if graph is weighted

type: dict(int : list)

vertex ↑ ↑ neighbors

Note. must have an empty list before appending

use defaultdict library:

```
from collections import defaultdict
defaultdict(dict)
```

Adjacency Matrix Representation:

```
initialize matrix[V][V]
for each edge:
    matrix[source][target] = 1
    matrix[target][source] = 1
```

both for undirected graphs

```
graph = [
    [0, 1, 0, 0, 1],
    [1, 0, 1, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 0, 1],
    [1, 1, 1, 0, 0]
]
```

size: $O(V^2)$

use # of weight here if graph is weighted

Graph Algorithms: use a frontier and visited DS
1. add first node to frontier
2. pop node and add all adjacent nodes to frontier
3. add popped node to visited list
4. repeat until frontier is empty

frontier: what are we visiting next?
visited: what have we already seen?

Breadth-First Search: O(V+E)
- frontier: queue
- visited set: list

Depth-First Search: O(V+E)
- frontier: stack
- visited set: list

Note: To catch cycles
- use a set for all visited lists
if vertex is in visited:
do not add it to frontier

Dijkstra's SSSP: O(V+ElogE)
- frontier: greedy algorithm by priority queue by min-heap
  - contains: (accumulated distance, vertex)
- visited: dictionary
  - contains: {vertex: total distance}

Note: use:
from collections import deque
import heapq

Dijkstra's SSSP storing path: same except,
~~frontier~~: (distance, vertex, source)
visited: {vertex: source}

Minimum Spanning Tree (MST): shortest possible tree that connects all nodes

Prim's MST Algorithm: O(V+E log E)
~~frontier~~: priority queue of min heap
- contains: [(distance, vertex, source), ...]
visited: dictionary
- contains: {vertex: source}

Note:
- dijkstra's is always in reference to starting node
- prim's starting node is arbitrary
ie. distances are not in reference to anything

Kahn's Topological Sort Algorithm: slightly different
create a dict[int, int] table of degrees for each vertex
while frontier (not empty):
pop vertex from frontier
append to visited
for each neighbor of vertex:
decrement degree
if degree == Ø:
add to frontier

frontier: set   time: O(V+E)
visited: list

detecting cycles:
if # visited < # vertices in graph:
there was a cycle

Programming Interviews: 12.15.24

Note. A computer can only run 10^8 iterations per second, and your code is expected to run in 1 second

* This can be used to your advantage *

Input Size (N)	Time Complexity	Approaches
$N \leq 11$	$O(N!)$	Permutations
$N \leq 26$	$O(c^N)$	Exploring all Subsets Recursive Traversals
$N \leq 100$	$O(N^4)$	Quadruple Loops
$N \leq 500$	$O(N^3)$	Triple Loops Brute Force Floyd-Warshall
$N \leq 10^4$	$O(N^2)$	Double Loops Simple Sorting
$N \leq 10^6$	$O(N \log N)$	Efficient Sorting Binary Search (N times) Heaps
$N \leq 10^8$	$O(N)$	Linear Traversals Hashing Counting
$N > 10^8$	$O(\log N), O(1)$	Binary Search Euclidean Algorithms Basic Math

Start with working inefficient solution:

- no partial credit
- code is never looked at

Common Patterns:

Is it a Graph? —yes→ Is it a tree? —yes→ DFS

Is it a tree? ↓ no → DAG? —yes→ Topological Sort

DAG? ↓ no → Shortest Path? → Weighted? —yes→ Dijkstra; —no→ BFS

Shortest Path? ↓ no → Connectivity? —yes→ Disjoint Set Union

Connectivity? ↓ no → Small Constraints? —yes→ DFS/backtracking; —no→ BFS

Is it a Graph? ↓ no → kth largest/smallest number? —yes→ Heap/Sort

kth largest/smallest number? ↓ no → Linked Lists? —yes→ 2 pointers

Linked Lists? ↓ no → Small Constraints? —yes→ Is Brute Force fast enough? —yes→ Brute Force/Backtracking; —no→ Dynamic Programming

Small Constraints? ↓ no → Subarrays/Substrings? —yes→ About Addition/Sums? —yes→ Prefix Sums; —no→ Sliding Window

Subarrays/Substrings? ↓ no → Max/Min? —yes→ Monotonic? —yes→ Binary Search

Monotonic? ↓ no → Can be split to Subproblems? —yes→ Dynamic Progra...; —no→ Greedy Alg

Max/Min? ↓ no → # of ways...? —yes→ Small Constraints?

# of ways...? ↓ no → Multiple Sequences? —yes→ Monotonic? —yes→ 2 pointers

Monotonic? ↓ no → Subproblems? —yes→ Dynamic Pro...

Multiple Sequences? ↓ no → Find/Enumerate Indices —yes→ Monotonic?; —no→ O(1) required

O(1) required —yes→ Monotonic?

Useful Python Libraries:

import collections

collections.deque

```
d=deque()      #init
append(x)      #useful methods
appendleft(x)
clear()
count(x)
extend(iterable)
extendleft(iterable)
insert(i, x)
pop()
popleft()
remove(value)
reverse()
rotate(n=1)  #rotates deque n steps to the right
             # can be negative
```

collections.defaultdict

```
d=defaultdict(list)  #typically used for graphs
```

```
import heapq
h=heapq.heapify(list)  #create a min heap
heappush(heap, item)   #useful methods
heappop(heap)
heappushpop(heap, item)
heapreplace(heap, item)
nlargest(n, iterable, key=None)
nsmallest(n, iterable, key=None)
```