alex-levesque.com

ELEC 278 - Lecture Notes

Fundamentals of Information Structures

Prof. Kexue Zhang • Fall 2025 • Queen's University

Contents

1 What is a Data Structure **4**

2 Pointer Arithmetic **5**
2.1 Indexing Pointers 5

3 Structures **7**
3.1 Typedef 8
3.2 Miscellaneous Struct Information 8
3.3 Dynamic Memory 10

4 Linked Lists **11**
4.1 Building a Linked List 11
4.2 Operations on Linked Lists 12

5 Double Linked Lists and Dynamic Arrays **20**
5.1 Double Linked List 20
5.2 Dynamic Arrays 20
5.3 Inserting an Element in the Middle 21
5.4 Deleting from the middle 22

6 Stack **24**
6.1 Modifying the top element 25
6.2 Implementing Stack using Linked List 26

7 Queue and Deque **28**
7.1 Implement Queue with Linked List 28
7.2 Implement Queue with Array 29
7.3 Dequeues and Operations on Queues 30

8 Recursion **33**

9 Algorithm Analysis **35**
9.1 Axioms 35
9.2 Examples 35

10 Big Oh Notation **36**
10.1 Time Functions 36

10.2 Properties of $O()$ . . . 36

11 Trees **38**
11.1 Terminology and Types of Trees . . . 40

12 Binary Tree Traversal **41**

13 Binary Search Trees **44**
13.1 Searching a BST . . . 44
13.2 Inserting a new value into a BST . . . 45
13.3 Deleting a Node from a BST . . . 46
13.3.1 Case 1: Node is a leaf . . . 47
13.3.2 Case 2: Target Node has One Child . . . 48
13.3.3 Case 3: Target Node has Two Children . . . 48
13.4 Finding Min & Max of a BST . . . 50

14 AVL Trees and Rotations **52**
14.1 Case 1: Left-Left or Right-Right . . . 54
14.2 Case 2: Left-Right or Right-Left . . . 55

15 Red Black Trees **59**

16 Heaps and Priority Queues **60**
16.1 Heaps . . . 60
16.1.1 Minheaps . . . 60
16.1.2 Maxheap . . . 61
16.1.3 Extract Item from Heap (using array) . . . 62
16.2 Stability . . . 63

17 Hash Tables **64**
17.1 Map Operations . . . 64
17.2 Hash Function and Collisions . . . 64
17.3 Closed Hash Table . . . 65
17.4 Open Hash Table . . . 65

18 Selection Sorting **68**
18.1 Simple Selection Sort . . . 68
18.2 Bubble Sort . . . 69
18.3 Heap Sort . . . 70

19 Binary Search **73**

20 Exchange Sorting **76**
20.1 Shell Sort . . . 76
20.2 Quick Sort . . . 78

21 Distribution Sorting **81**
21.1 Bucket Sort . . . 81
21.2 Radix Sort . . . 82

22 Merge Sorting **84**
22.1 Divide and Conquer 84
22.1.1 Merging Process 84
22.1.2 Partition Process 85

23 Graphs **87**
23.1 Types of Graphs 87
23.2 Connectivity 88
23.3 Representation 89

24 Graph Traversals **91**
24.1 Breadth-First Search 91
24.1.1 BFS Running Time 92
24.2 Depth-First Search 93
24.2.1 DFS Running Time 95

25 Shortest Path Problems **96**
25.1 Dijkstra's Algorithm 96
25.2 Subgraph and Spanning Tree 98
25.3 Prim's Algorithm 99

26 ELEC 278 Final Study **102**

1 What is a Data Structure

A data structure is a way to store and organize data in order to facilitate access and modification

It concerns the **representation, manipulation, and efficient management of data**. Use cases involve large databases and internet indexing services.

Efficient data structures are key to designing efficient algorithms.

Each data structure supports one or more algorithms for the **operations**, that are insert, delete, search, and modify

Conversely, an algorithm is a step by step procedure in performing a task

2 Pointer Arithmetic

Incorrect swap function

```
void swap(int a, int b){
    int tmp = a;
    a = b;
    b = tmp;
    }

int x = 1;
int y = 2;
swap(x,y);
```

This function is swapping its own copy of the data, and these changes are not reflected outside the function Pointers are necessary to affect the original variables

Pointers

```
int *nptr = &total;
```

This pointer is pointing to the memory address of the total variable

The & operator returns or takes the address of a variable

The * operator uses or dereferences the value of the pointer

Usage:

```
int total = 5;
float speed = 2.3;
int numbers[4];
int *nptr = &total;
*nptr = 12; # this changes 5 to 12
```

Correct swap function

```
void swap(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = *tmp;
    }
int x = 1;
int y = 2;
swap(&x, &y); # x and y are changed
```

Pointer values are copied, and pointers are used to access original values

2.1 Indexing Pointers

Neglecting [] of an array calls its first value (at index 0)

```
int numbers[4];
int *nptr = numbers;
```

Notice, no & for pointing to arrays

You can add or subtract integers from pointers. When you do this, the pointer's value is scaled by the size of its referenced type (e.g. nptr + 2 is equivalent to adding 8 bytes if pointing to int since each int is 4 bytes)

You can access array elements using pointer arithmetic:

```
*(nptr+2)=10; # sets the third element of the array to 10
```

3 Structures

A data structure is a way to store and organize data in order to facilitate access and modification

Definition: A struct groups items of possibly different types into a single type

```
struct structure_name{
    data_type member_name1;
    data_type member_name2;
    };
```

The items in the structure are called **members** or **fields**

We can define structure variables as

```
struct structure_name variable_name1;
```

or at the same time:

```
struct structure_name{
    data_type member_name1;
    data_type member_name2;
    } variable_name1;
```

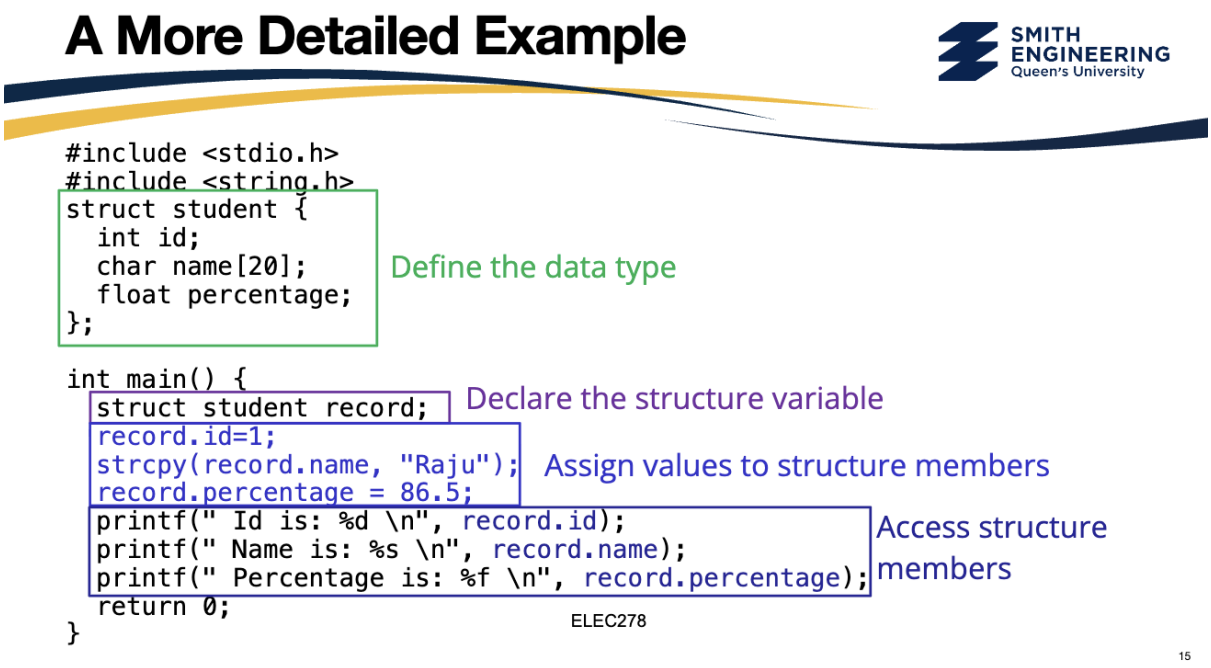
Where *structure_ name* is the defined structure of data, and *variable_ name* is an instance of the struct

Use the dot (.) operator to access members

```
var_name.marital_status = 4;
```

and initialize individual fields with the dot notation:

```
struct Student student1={.name = "Alice", .section = 'A'};
```



3.1 Typedef

A typedef looks like a variable definition, but defines a new name for an existing type. This can improve code readability.

```
typedef int studentNumberType;
studentNumberType studentNumber1;
```

Here, studentNumberType becomes an alias for int, so studentNumber1 is actually an int but is made clearer.

```
typedef struct {
    char name[50];
    int class;
    char section;
    } Student;
Student s1, s2;
```

3.2 Miscellaneous Struct Information

If we typedef a struct, and use it in another structure, one of the members of a structure is itself another structure

```
typedef struct {
    int imag;
    float real;
} complex;

struct number {
    int flags;
    complex phase;
} num1, num2;
```

Where *phase* is a *struct*

C Enumerated Types

This is a way to create a user-defined type consisting of a set of named integer constants

```
enum colors {RED, GREEN, BLUE}
```

You can also change the starting value by:

```
enum week {Monday = 1; ...} # Sunday is 0
```

We can also combine enums with typedef

Structs can be used as the base type of an array

```
typedef struct {
    int x;
    int y;
} point;

point vertexes[100];

vertexes[4].x=23;
vertexes[4].x=18;
```

We can also point to members within a struct

```
typedef struct {
    int x;
    int y;
} point;

point a = {23,18};
point *b = &a;
(*b).x=34;
```

We can also return structs as a result of a function

Pointers to pointers are used to reference the address of another pointer

```
void allocateInt(int **p) {
    *p = malloc(sizeof(int))
}
```

or

```
void swap(point **a, point **b){
    point = *tmp;
    tmp = *a;
    *a = *b;
    *b = *tmp;
}
```

3.3 Dynamic Memory

So far, memory was either global or local. Now we add **Heap memory**, to request memory at runtime

```
tax_info *bob = (tax_info*) malloc(sizeof(tax_info));
bab->martial_status = single;
```

The arrow operator is shorthand for (*ptr).field

Instead of storing 100 structs (3200 bytes), you can store 100 pointers (800 bytes) and allocate each struct individually:

```
tax_info *employees[100];
employees[0] = malloc(sizeof(tax_info));
```

To duplicate a string dynamically:

```
char *s1 = "Hello";
char *s2 = strdup(s1); # strdup is like copying into another string
```

Since functions also live in memory, we can point to them too:

```
int sum(int a, int b) { return a+b; }
int (*func_ptr)(int,int) = sum;
printf("%d\n", func_ptr(2,3));
```

4 Linked Lists

Use structure to group array and count of number elements in the array * Arrays are declared with a maximum length

Linked lists provide an ordered collection that grows with the number of data items * each element is a node with data and a pointer to the next element * Head points to the first element

Definition: Linked lists are a means of dynamically allocating an ordered collection of elements

Why?: Arrays are rigid. If you don't know how many items you'll need, arrays can overflow. Linked lists solve this with dynamic allocation.

Each element (**a node**) stores: * Data * Pointer to the next node

The list is ended with a *NULL* ptr

```
struct node {
    int value;
    struct node *next;
};
struct node *head; // points to first node
```



4.1 Building a Linked List

To begin building a linked list, memory is first allocated for the head node using `malloc`, and its value field is set to `14`

```
head = (struct node *) malloc (sizeof(struct node));
head -> value = 14;
```

Next, another node is added by allocating memory for `head->next`, and its value is set to `92`

```
head -> next = (struct node *) malloc (sizeof(struct node));
head -> next -> value = 92;
```

After allocating each node, it is important to check if memory allocation succeeded; if `head` is `NULL`, the program prints "failed" and exits

```
if (head == NULL) {
    printf("failed");
    exit(1);
};
```

We can use linked lists with more complex datatypes:

Example:

```
typedef struct {
    float x;
    float y;
    int alt;
} aircraftPos;
```

We can build a node struct, which is the building block of the linked list, where each node contains a value of type aircraftPos and a pointer to the next node on the list

```
typedef struct node {
    aircraftPos value;
    struct node *next;
} node;
```

4.2 Operations on Linked Lists

Traversing

Start function

```
void printList(struct node *head) {
```

To traverse a linked list, you begin by pointing a temporary node `tmp` to the head, then print an opening bracket to start the list display

```
struct node *tmp = head;
printf("[ ");
```

The traversal continues in a loop while `tmp` is not `NULL`; this ensures you stop once you reach the end of the list

```
while (tmp != NULL) {
```

Inside the loop, you first access the current node's value by printing it, and then move forward by updating `tmp` to its `next` pointer

```
    printf("%d, ", tmp->value);
    tmp = tmp->next;
}
```

Finally, you print a closing bracket and a newline to finish displaying the entire list

```
print("]\n");
```

Adding a node to the list

To simplify creating nodes, we can define a helper function `newNode` that allocates memory with `malloc`, assigns the given value, initializes the `next` pointer to `NULL`, and returns the new node

```
node *newNode(int value){
    node *tmp = (node *)malloc(sizeof(node))
    tmp -> value = value;
    tmp -> next = NULL;
    return tmp;
}
```

To add a new node to an empty list, we call `newNode` to create a node with value `14`, and if the head is `NULL` (meaning the list is empty), we set `head` to point to this new node

```
new = newNode(14);
if (head != NULL) {
    head = new;
}
```

We can also insert at the start of an existing list by creating a new node, setting its `next` pointer to the current head, and then updating head to point to this new node

```
node *new = newNode(12);
new->next = *h;
*h = new;
```

This insertion-at-front logic can also be written as a reusable function, `insertFirst`, which takes a double pointer to the head, creates a new node, links it before the current head, and updates the head pointer

```
void insertFirst(node **h, int data){
    node *new = newNode(data);
    new -> next = *h;
    *h = new;
}

insertFirst(&head,14);
```

Appending to an existing list

Start function

```
void appendList(node **h, int data){
  node *new = newNode(data);
```

Cover empty list case:

```
if (*h == NULL) { *h = new; return; }
```

To append to an existing linked list, we first set a pointer `tail` to the head of the list, and then create a new node with the desired value using `newNode`

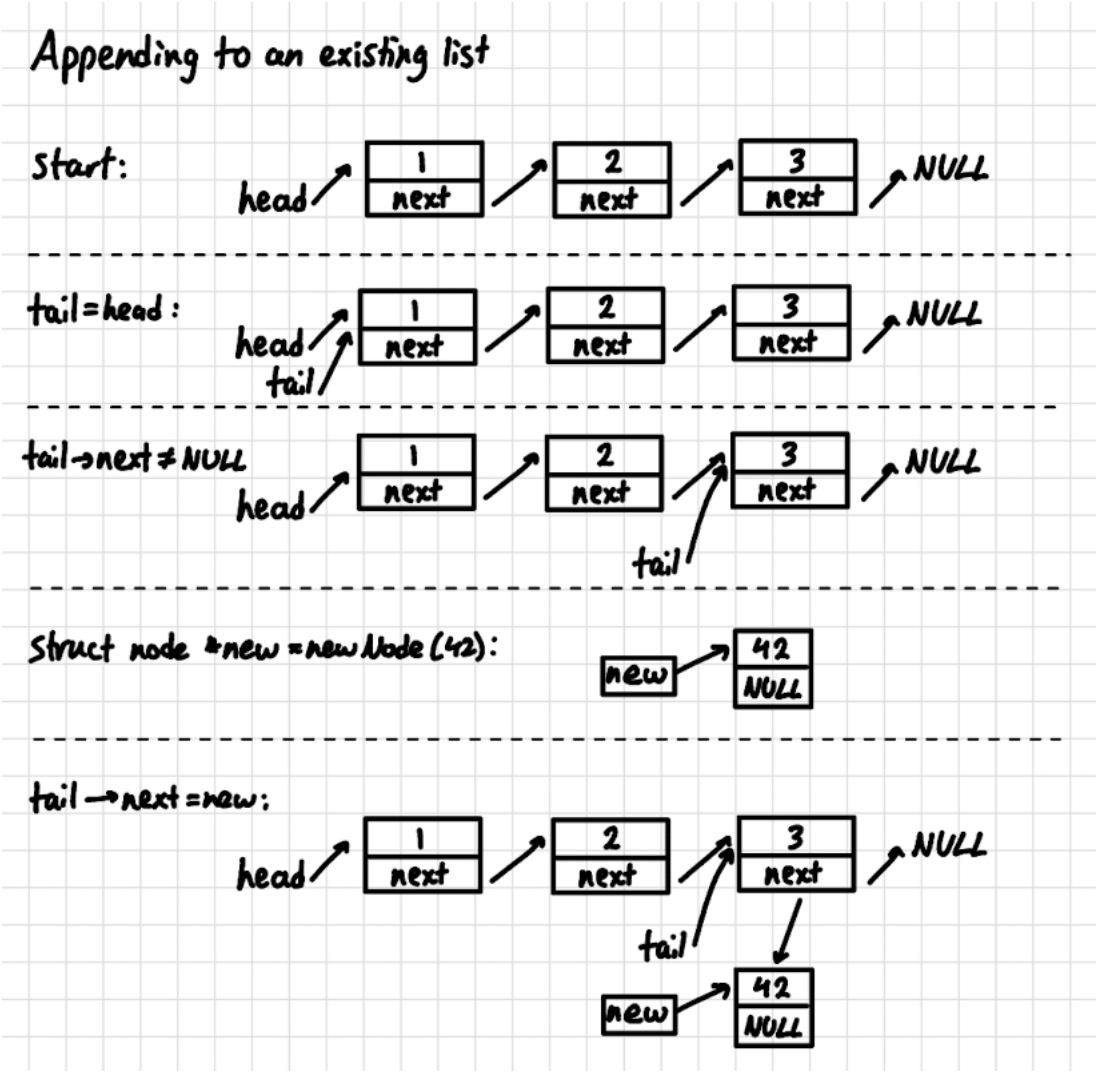
```
node *tail = *h; // start at the head
struct node *new = newNode(42);
```

Next, we traverse the list by moving `tail` forward until we reach the last node, which is identified when `tail->next` becomes `NULL`

```
while (tail -> next != NULL) {
    tail = tail -> next; // traverse to the last node
    }
```

Finally, once at the last node, we attach the new node by setting `tail->next` to point to it, completing the append operation

```
tail -> next = new; // link the new node at the end
```



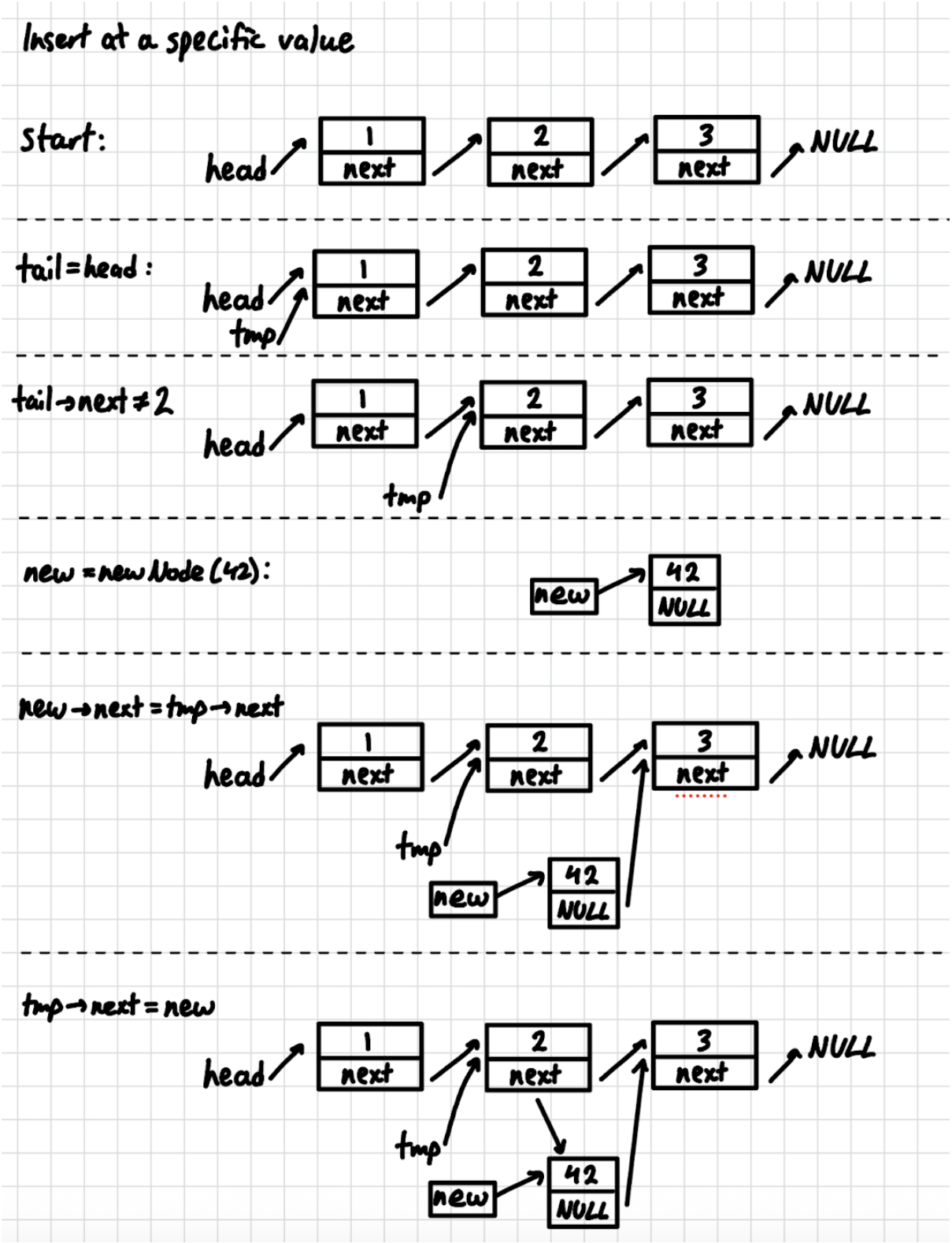
Insert a Value in the Middle of a List (after a specific value)

To insert a new node after a specific value, we first create a temporary pointer `tmp` starting at the head, and then traverse forward while `tmp` is not `NULL` and the current node's value is not equal to `92`

```
struct node *tmp = head; // start at head
while (tmp!=NULL && tmp->value!= 92){
  tmp = tmp->next;  // move to the next node
}
```

If the loop finds a node with value `92`, then `tmp` points to the node. At this point, we create a new node with value `42`, set its `next` pointer to the node after `tmp`, and then adjust `tmp->next` so that it links to the new node

```
if (tmp != NULL){
  new = newNode(42); // create node with value 42
  new -> next = tmp -> next; // link new node to the node after tmp
  tmp -> next = new; // insert new node right after tmp
}
```



Insert Sorted Function

The function starts by creating a new node with the given value using `newNode`

```
node *new = newNode(value);
```

If the list is empty, the new node becomes the head, and the function immediately returns

```
if (*head == NULL){            // Case 1: Empty list
    *head = new;
    return;
}
```

If the new value is smaller than the current head's value, the new node is inserted at the front, and head is updated

```
if ((*head)->value > value){   // Case 2: Insert before the first node
    new->next = *head;
    *head = new;
    return;
}
```

Otherwise, the function traverses the list using a pointer `tmp`, always looking one node ahead. If it finds that the next node's value is larger than the new value, it inserts the new node between `tmp` and `tmp->next`

```
node * tmp = *head;
while (tmp->next != NULL){              // Traverse the list
    if (tmp->next->value > value){      // Case 3: Insert in middle
        new->next = tmp->next;          // Point new node to next node
        tmp->next = new;                // Link previous node to new
↪   node
        return;
    }
    tmp = tmp->next;                    // Advance to next node
}
```

If no larger value is found, the function appends the new node at the end of the list

```
tmp->next = new; // Case 4: Insert at the end
```

Removing Elements

Often, we want to remove the first node, a specific node, or the last node

Delete First Node

The function `removeFirst` begins by saving a pointer `tmp` to the current head of the list and initializing an integer variable `value` to `0`, which acts as a default return value

```
struct node *tmp = *head;   // Save pointer to current head
int value = 0;              // Default return value if list is empty
```

It then checks if the list is non-empty, if so, the head pointer is advanced to the second node, the value of the original head is saved into `value`, and the memory allocated to the old head node is freed

```
if (*head != NULL){
    *head = (*head)->next;  // Move head to next node
    value = tmp->value;     // Save the value of the old head
    free(tmp);              // Free memory of the old head
}
```

Finally, the function returns the integer `value`, which is either the removed node's value or `0` if the list was empty

```
return value'
```

Delete Specific Value

The function `deleteValue` starts by checking if the head of the list is `NULL`, and if so, it immediately returns because there is nothing to delete

```
if (*head == NULL) return;
```

Next, it handles the special case where the first node contains the target value: it saves a pointer to the head in `tmp`, updates `head` to the second node, frees the old head, and then returns.

```
if ((*head)->value == val) {
  node * tmp = *head;
  *head = (*head)->next;
  free(tmp);
  return;
}
```

If the value is not in the head, the function creates a pointer `prev` starting at the head, and then iterates while `prev->next` is not `NULL`

```
node * prev = *head;
while (prev -> next != NULL) {
```

Inside the loop, it checks if the next node holds the target value; if so, it saves that node in `tmp`, bypasses it by updating `prev->next`, frees the node, and returns.

```
  if (prev -> next -> value == val){
    node * tmp = prev -> next;
    prev -> next = tmp -> next;
    free(tmp);
    return;
  }
```

If the value was not found, the loop moves `prev` to the next node and continues searching until the end of the list.

```
prev = prev->next;
```

5 Double Linked Lists and Dynamic Arrays

5.1 Double Linked List

A single linked list has a single entry point (head)

Idea: Each node has `next` and `prev`, plus you often keep `head` and `tail` pointers so you can traverse both ways.

Hence a Double Linked List, where one linked list goes forward and one goes backwards

```
typedef struct node {
  int value;              // payload
  struct node * next;     // forward link
  struct node * prev;     // backward link
} node;

node * head;              // first node pointer
node * tail;             // last node pointer
```

5.2 Dynamic Arrays

Idea: Wrap an array with a count field so you know how many elements are "live"

```
typedef struct {
    int *array;
    size_t capacity; // available space
    size_t length; // nmber of elements stored in the array
} Array;
```

Initialization

Allocate an initial buffer, starting at length 0 and capacity at initial size

```
void initArray(Array *a, int initialSize){
  a->array = (int *)malloc(initialSize * sizeof(int)); // allocate
↪  buffer
  a->length = 0;                      // nothing used yet
  a->capacity = initialSize;          // remember capacity
}
```

Append an Element to the Array

```
void addToArray(Array *a, int element){
```

The function begins by checking if the array is already full. It compares the number of used entries (`length`) to the total slots available (`capacity`). If they are equal, the array must grow.

```
if (a->length == a->capacity){
```

In that case, it doubles the capacity and resizes the buffer with `realloc`, ensuring that all existing elements are preserved while making space for new ones

```
a->capacity *= 2;
a->array = (int*)realloc(a->array, a->capacity * sizeof(int));
}
```

After ensuring enough capacity, the function places the new element at the current end of the array (`a->array[a->length]`), then increments `length` so the array correctly reflects its new size

```
a->array[a->length++] = element;
```

And free the dynamic array

```
void freeArray(Array *a){
  free(a->array);                  // release heap buffer
  a->array = NULL;               // null out dangling pointer
  a->capacity = a->length = 0;     // reset counters
}
```

Note: Arrays are contiguous memory. Inserting an element into the memory of the list means moving the remaining elements up one position

We use `memove` to copy `n` bytes from the memory pointed to by `src`, to the memory pointed to by `dst`. This function understands memory blocks overlapping

```
void * memove(void *dst, const void *src, size_t n)
```

5.3 Inserting an Element in the Middle

```
bool insertAt(Array *a, size_t index, int value){
```

The function begins by validating the requested index. If the index is greater than `length`, the function returns `false`

```
if (index > a->length)    // inserting past the end is invalid
  return false;
```

Next, it checks if the array is already full. If `length == capacity`, the capacity is doubled, and the buffer is resized using `realloc`. This ensures there is enough space for the new element

```
if (a->length == a->capacity){
  a->capacity *= 2;
  a->array = realloc(a->array, a->capacity * sizeof(int));
}
```

To make room for the new element, all elements from `index` up to the last used position are shifted one slot to the right. This is efficiently handled with `memmove`, which safely copies overlapping memory regions

```
memmove(&a->array[index + 1],        // shift destination
        &a->array[index],            // shift source
        (a->length - index) * sizeof(int); // number of bytes to
        ↪  shift
```

Finally, the new value is written into the open slot at `index`, and `length` is incremented to account for the insertion. The function returns `true` to indicate success

```
a->array[index] = value;
a->length++;
return true;
```

5.4 Deleting from the middle

The function begins by validating the index.

```
if (index >= a->length)    // invalid index (out of range)
  return false;
```

If the index is valid, the function reduces the logical size of the array by decrementing `length`. This means the element at the given index will effectively be removed

```
a->length--;
```

To fill the gap, all elements after the removed one are shifted left by one slot. `memmove` is used here because it safely handles overlapping regions of memory

```
memmove(&a->array[index],        // overwrite deleted slot
        &a->array[index + 1],    // start from next element
        (a->length - index) * sizeof(int)); // shift the suffix left
```

After the shift, the function optionally applies a shrink policy. If the number of elements is at or below one-quarter of the current capacity, the capacity is halved and the buffer resized with `realloc`

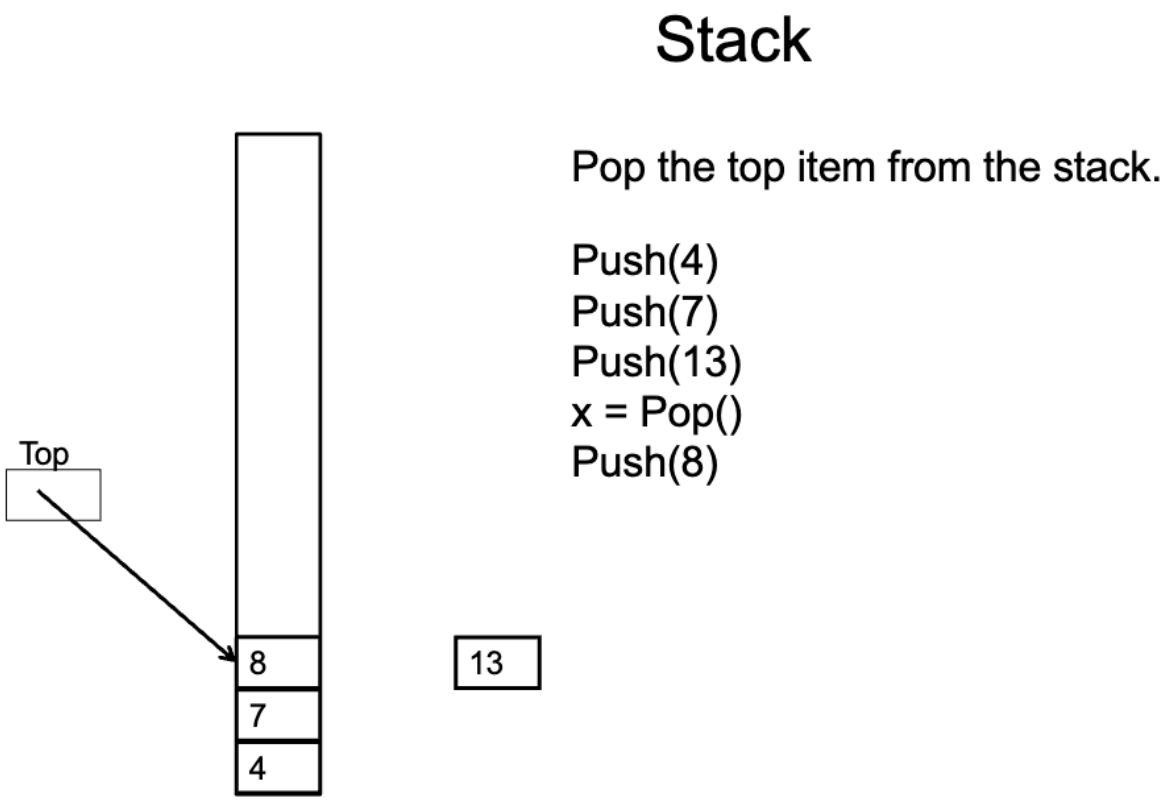
```
if (a->length * 4 <= a->capacity){
  a->capacity /= 2;
  a->array = realloc(a->array, a->capacity * sizeof(int));
}
return true;
```

6 Stack

Definition: A data structure for storing a collection of data items, where items can be added to and removed from the collection, but *only the last item added to the stack can be accessed or removed.* This is a last-in, first-out (LIFO) data structure

Common Functions:

`push(item)` puts an item on the top of the stack `item=pop()` retrieves the top item from the stack `peek()` retrieves the top item of the stack without removing it



LIFO is important for use cases such as Reversing a word, web browsers to store addresses of recently visited sites, and undo functions in applications

Implementation requirements

Top variable always indicates to the top element of the stack - expect when the stack is empty - special value to indicate the stack is empty

Some mechanism to indicate errors: - attempt to pop an empty stack - out of memory when pushing an element

Example:

`typedef struct { ... } ArrayStk` defines a new struct type for a stack with an alias, holds an index of the current top element in the stack `top`, and sets an array `data` of size 100 to store the stack's elements

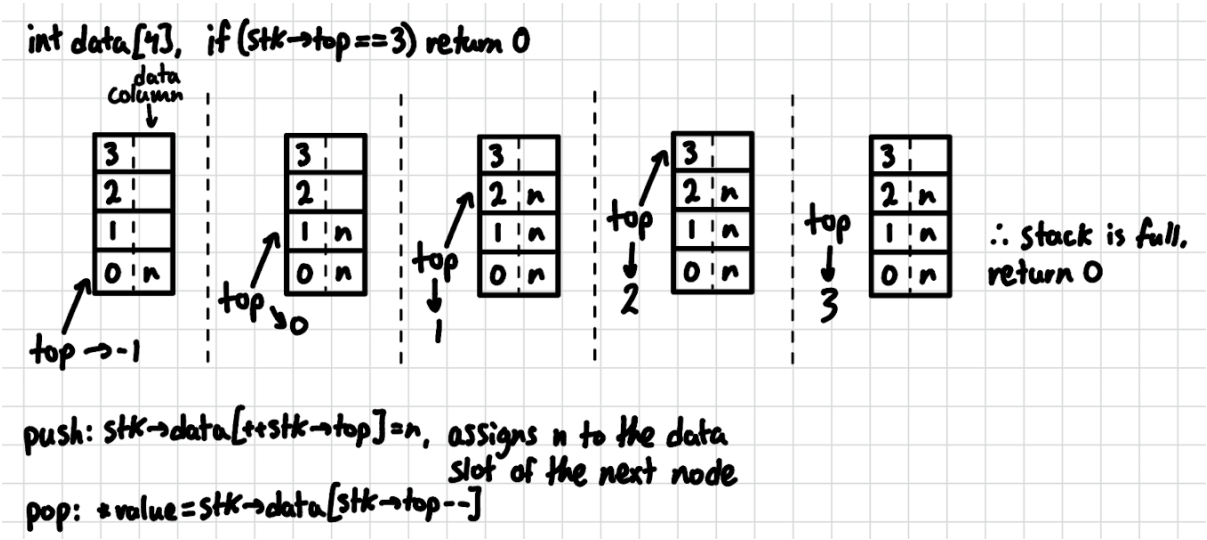
```c
typedef struct {
  int top;                // index of current top, -1 when empty
  int data[4];           // storage
} ArrayStk;
```

This function `push` tries to insert a value `n` onto the stack, and returns 1 if successful and 0 if not. The function also pre-increments `stk->top` (moves it up one position) and stores the new value `n` in that position in the `data` array.

```c
int push(ArrayStk *stk, int n){
  if (stk->top == 3) return 0;          // full
  stk->data[++stk->top] = n;             // push
  return 1;
}
```

The `pop` function checks if full, stores it in that variable pointed to by `value`, then post-decrements `stk->top` (moves it down one position), effectively removing it from the stack

```c
int pop(ArrayStk *stk, int *value){
  if (stk->top == -1) return 0;          // empty
  *value = stk->data[stk->top--];        // pop
  return 1;
}
```



6.1 Modifying the top element

Setup:

```c
ArrayStk stk = { .top = -1 };    // empty stack
int i, value, *ptr_top;          // local vars
```

Pushes 20, 21, 22, 23, 24 onto the stack. Then pops them all off, printing in *reverse order* because of LIFO

```
for (i=0; i<5; i++) { push(&stk, 20+i); }
while (pop(&stk, &value)) {
  printf("%d\n", value);
}
```

This pushes 100, 101, 102, 103, 104

```
for (i=0; i<5; i++) { push(&stk, 100+i); }
```

Pops the top into `value`, calls `tos` to return a pointer to the new top element after the pop, then adds 50 to that element in place

pops 104 153 152 151 150

```
while (pop(&stk, &value)) {
  printf(" %d", value);
  if (tos(stk, &ptr_top))
    *ptr_top = *ptr_top + 50;
}
```

where `tos` is:

```
int tos(ArrayStk stk)(int **ptop) {
    if (stk.top == -1) return 0;
    *ptop = &stk.data[stk.top];
    return 1;
}
```

6.2 Implementing Stack using Linked List

Building the stack data structure on top of a linked list instead of using an array:

This defines a struct with an alias, and a pointer to the next node in the stack (meaning the node below the top). This allows us to chain nodes like a linked list. We also have `nodval`, which is the actual data being stored in the stack

```
typedef struct _stk_node {
  struct _stk_node *next;   // link to previous top
  int nodval;               // payload
} StkNode;
```

- Function takes a pointer to the stack head pointer to modify the pointer outside the function
- Allocate memory for a new node on the heap, and stores the new value inside the node

- Links the new node to the current stack and updates the head pointer so the new node is now the stack's top.

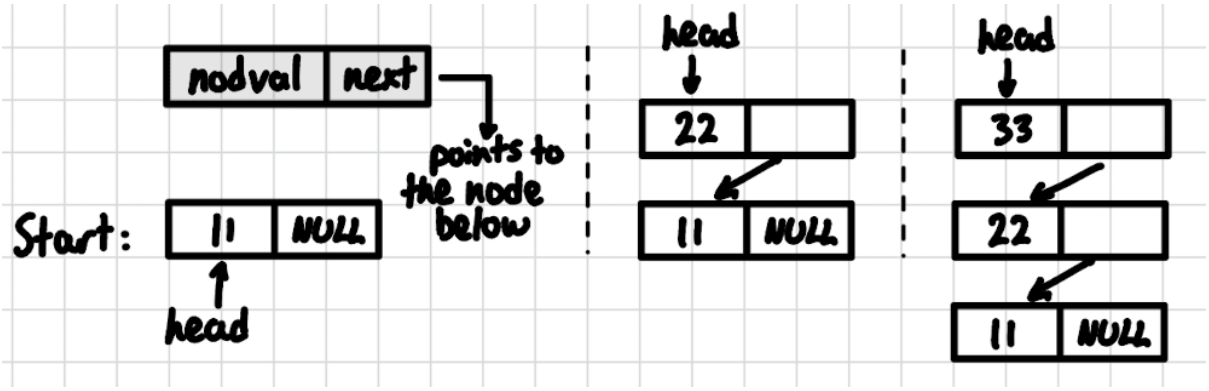
```
void push(StkNode **stkHead, int n){
  StkNode *p = malloc(sizeof *p);  // allocate new node
  p->nodval = n;                   // set payload
  p->next = *stkHead;              // chain to old top
  *stkHead = p;                    // new node becomes top
}
```

This pop function removes the top node, and returns 0 if the stack is empty (checks for underflow). It firstly saves the current top node pointer, moves the stack head down to the next node, and releases memory for the removed node.

```
int pop(StkNode **stkHead, int *out){ // (slide shows int return; safer
↪  to output)
  if (*stkHead == NULL) return 0;     // underflow guard
  StkNode *p = *stkHead;              // old top
  *out = p->nodval;                   // capture value
  *stkHead = p->next;                 // drop node from stack
  free(p);                            // free storage
  return 1;                           // success
}
```

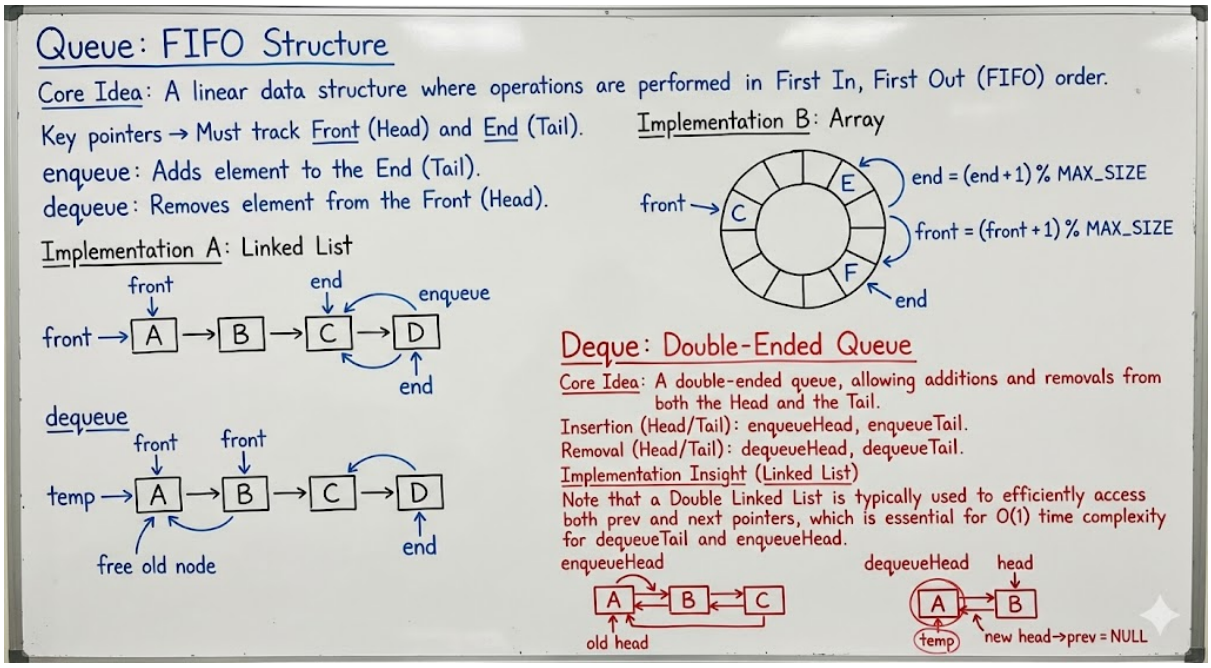
Top-of-stack returns the address of `nodval` to the head node, else return NULL

```
int *tos(StkNode *stkHead){
  return stkHead ? &stkHead->nodval : NULL;  // pointer to top value or
  ↪  NULL
}
```



7 Queue and Deque

- A linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO)
- Operations: `enqueue` adds an element to the end, `dequeue` removes an element from the front, `isEmpty`
- We need to track the Front (head) and End (tail), thus we need a pointer to the end and front



7.1 Implement Queue with Linked List

Enqueue inserts a node at the rear (end) of the queue

```
void enqueue (int n) {
```

Allocate memory for a new node, assign value and next pointer to NULL

```
struct item *pnew = malloc(sizeof(struct item));
pnew->value = n;
pnew->next = NULL;
```

Nudge n to `pnew`

```
if (end != NULL){
end->next = pnew;
}
if (front == NULL){ // if front is NULL, the list is empty and set
 ↪  front to new
front = pnew;
}
count++;
```

Dequeue removes an element from the front (head) of the queue

```
bool dequeue (int *n){
```

Set new node `temp` to front Give the pointer value `*n` to front

```
struct item *temp = front;
*n = (front -> value);
```

Move `front` to the next node

```
front = front-> next;
```

Free memory of old front (temp)

```
free(temp);
count--;
return true;
```

7.2 Implement Queue with Array

Common way to implement queue with array is with a circular model. This fixes the idea of a fixed-size array

```
# initialize
// define max size
// define queue array
// define count, front, end = 0
// define isEmpty and isFull functions
```

```
bool enqueue(int n){
// if full, return false
// add to end of queue
end = (end+1) % MAX_SIZE;
count++;
return true;
}
```

```
bool dequeue(int *n){
// if empty, return false
// remove from front of queue
front = (front + 1) % MAX_SIZE;
count--;
return true;
}
```

7.3 Dequeues and Operations on Queues

- a double-ended queue, can be done with either an array or double linked list
- Operations: `enqueueHead`, `enqueueTail`, `dequeueHead`, `dequeueTail`, `getHead`, and `getTail`

EnqueueHead inserts a new node before the head

```
void enqueueHead (int n) {
struct item *pnew = malloc(sizeof(struct item)); // allocate memory for
 ↪  node

pnew -> value = n;
pnew -> next = NULL;
pnew -> prev = NULL; // assign value and pointers

// update head node
if (head != NULL){
    head -> prev = pnew;
    pnew -> next = head;
}
head = pnew;

// if queue is empty, set tail
if (tail == NULL){
    tail = pnew;
}
count++;
}
```

EnqueueTail inserts a new node after the tail

```
void enqueueTail (int n){
struct item *pnew = malloc(sizeof(struct item)); // allocate memory for
↪  node

pnew -> value = n;
pnew -> next = NULL;
pnew -> prev = NULL; // assign value and pointers

// update tail node
if (tail != NULL){
pnew -> prev = tail;
tail -> next = pnew;
}
tail = pnew;

// if queue is empty, set front node
if (head == NULL){
head = pnew;
}
count++:
}
```

DequeueHead removes a node from the head

```
bool dequeueHead(int *n){
struct item *temp = head; // create temp pointer to head

if (count == 0) return false; // queue is empty, can't remove anything

*n = head -> value; // store head value to n

if (head -> next){
head -> next -> prev = NULL; // detach old head
head = head->next; // move head to next node
}

if (head==NULL) tail = NULL; // if head is NULL, queue is empty

free(temp); // deallocate old head node
count--;
return true;
}
```

DequeueTail removes a node from the tail

```
bool dequeueTail (int *n){
struct item *temp = tail; // create temp pointer to head

if (count == 0) return false; // queue is empty, can't remove anything

*n = tail -> value; // store head value to n

if (tail -> next){
tail -> next -> prev = NULL; // detach old head
tail = tail->next; // move head to next node
}

if (tail==NULL) head = NULL; // if head is NULL, queue is empty

free(temp); // deallocate old head node
count--;
return true;
}
```

8 Recursion

Sometimes a problem can be solved by first solving a smaller version of the same problem. Recursion means to define something in terms of itself

When the problem is small enough, then it can be solved directly, called the *base case*

Example:

Base Case: $1! = 1$

```
if (n==1){
temp=1;
}
```

Recursive Case $n! = n * (n-1)!$

Iterative vs. Recursive:

Iterative

```
def fact(n){
    result = 1;
    for i in range(1,n+1):
        result *= i;
    return result;
}
print(fact(i))
```

Recursive

```
def fact(n){
    if n==0 or n==1: // Base Case
        return 1;
    else:
        return n * fact(n-1); // Recursive Call
}
print(fact(n))
```

Linked List - Recursive Insert Sorted

newNode function

```
node *newNode(int value, node *next){
    node *tmp = malloc(sizeof(node));
    tmp -> value = value;
    tmp -> next = next;
    return tmp;
}
```

Insert function

```c
int insert(struct node **list, int value){

    // if list empty, or the current node's value is greater than the
    ↪  value to insert, create new node
    if (*list = NULL || (*list)-> value > value){
        *list = newNode(value, *list);
        return 1;
    }

    // if new value is smaller than current, insert it before the
    ↪  current node
    } else if ((*list)->value == value) {
        return 0;

    // otherwise, move one step deeper and try to insert (recursive
    ↪  step)
    else {
        return insert(&((*list)->next), value);
    }
    return 1;
}

int main(){
    struct node *head = NULL;
    insert(&head, 27);  // head: 27
    insert(&head, 92);  // head: 27 -> 92
    insert(&head, 12);  // head: 12 -> 27 -> 92
    insert(&head, 14);  // head: 12 -> 14 -> 27 -> 92
}
```

9 Algorithm Analysis

Algorithm analysis is about measuring how much computing resources (like time or memory) an operation or algorithm uses

9.1 Axioms

1. Fetching or storing an integer from memory takes a constant time. $y = x + 1$ takes $T_{fetch} + T_{store}$
2. Basic operations (add, subtract, multiply, divide, compare) on integers all take constant time. $y = y + 1$ takes $2T_{fetch} + T_{op} + T_{store}$
3. Function call and return times are constant (T_{call}, T_{return}), passing an integer argument is like fetching it. $y = f(x)$ takes $T_{fetch} + T_{call} + T_{f(x)} + T_{store}$
4. Array subscripting address calculation is constant T, not including time to compute i or fetch/store the element. $y = a[i]$ takes $3T_{fetch} + T_{store} + T$
5. Allocating memory is constant time T_{new}, not including initialization

9.2 Examples

- Finding the largest element in an array $\max_{0 \le i < n} a_i$

```
1 int FindMaximum(int a [], int n)
2 {
3         int result = a[0];
4         for(int i = 1; i < n; ++i)
5                 if(a[i] > result)
6                         result = a[i];
7         return result;
8 }
```

Statement	Time
3	$3T_{fetch} + T_{[.]} + T_{store}$
4a	$T_{fetch} + T_{store}$
4b	$(2T_{fetch} + T_{<}) \times n$
4c	$(2T_{fetch} + T_{+} + T_{store}) \times (n-1)$
5	$(4T_{fetch} + T_{[.]} + T_{<}) \times (n-1)$
6	$(3T_{fetch} + T_{[.]} + T_{store}) \times ?$
7	$T_{fetch} + T_{store}$

Line 6 executed only if

$$a_i > (max_{0 \le j < i} a_j)$$

? - depends on the actual elements of the array, $a_0, a_1, ..., a_{n-1}$

$T(n, a_0, a_1, ..., a_{n-1}) = t_1 + t_2 n + \sum_{i=1}^{n-1} t_3$ (with $a_i > (max_{0 \le j < i} a_j)$)

10

$t_1 = 2T_{store} - T_{fetch} - T_{+} - T_{<}$

$t_2 = 8T_{fetch} + 2T_{<} + T_{[.]} + T_{+} + T_{store}$

$t_3 = 3T_{fetch} + T_{[.]} + T_{store}$

$T_{average}(n) = t_1 + t_2 n + t_3 \Sigma_{i+1}^{n-1} \frac{1}{i+1}$

The probability that a_i is the largest of the $i + 1$ values, which is $\frac{1}{i+1}$ from $p_i = P[a_i > (max_{0 \le j < i} a_j)]$

$T_{worsecase}(n) = (t_1 - t_3) + (t_2 + t_3) \times n$ $T_{bestcase}(n) = t_1 + t_2 n + \Sigma_{i=1}^{n-1} p_i t_3$ when $p_i = 0$, thus $= t_1 + t_2 n$

Instead of detailing each parameter, we let T be the *clock cycle* of a machine, where $T_{fetch} = kt$

Assume that all timing parameters expressed in units of clock cycles. Thus $T = 1$. k is assumed to be the same for all parameters, thus $k = 1$

10 Big Oh Notation

10.1 Time Functions

The time function $T(n)$ describes how the running time of an algorithm grows as a function of the input size n

$f(n) = O(g(n))$ means for large enough n, $f(n)$ is at most a constant times $g(n)$

Example: Given $8n + 128$, show that $f(n) = O(n^2)$, find constants $n_0 > 0$ and $c > 0$ such that $\forall n \geq n_0, f(n) \leq cn^2$

If $c = 1$, then $f(n) \leq cn^2 \Rightarrow 8n + 128 \leq n^2 \Rightarrow 0 \leq (n - 16)(n + 8)$

Since $(n + 8) > 0$ for all values $n \geq 0$, then $(n_0 - 16) \geq 0, i.e. n_0 = 16$

Possible solution: for $c = 1, n_0 = 16, f(n) \leq cn^2$ for all integers $n \geq n_0$, hence $f(n) =)(n^2)$

10.2 Properties of $O()$

If two functions have the same $O(g(n))$, the functions are not necessarily equal, they just share the same upper bound on their growth

When adding functions, we take $f_1(n) + f_2(n) = O(max(g_1(n), g_2(n)))$

If we multiply two functions, we get $O(n^2) + O(n^3) = O(n^5)$

If we have a polynomial time or space functions, we take the term with the highest order without the constant, e.g. $f(n) = O(n^m)$

Big O: upper bound (worst-case growth) Big Ω: lower bound (best-case growth) Big Θ: tight bound (exact growth)

Sum Example

This algorithm sums elements in an array, let $n = 5$

```
int sum(a,n){
    s=0;
    for(int i = 0; i<n; i++){
        s=s+a[i];
    }
    return s;
}
```

frequency count method:

```
// 1
// for (1, n+1, n), use the greatest count (n+1)
// n
// 1

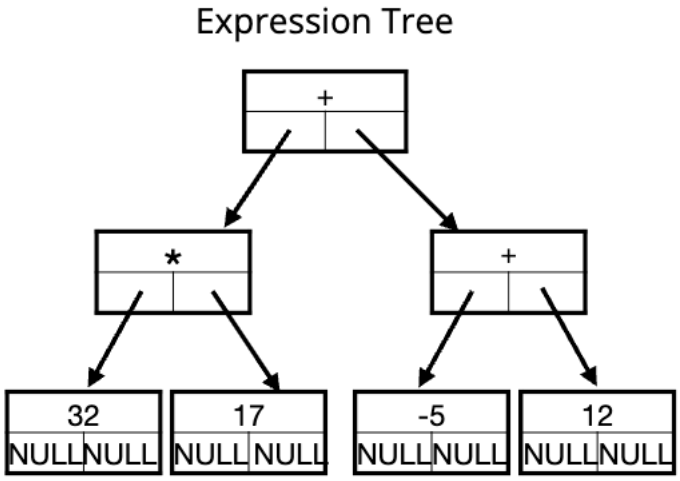
// time function f(n) = 2n+3
// O(n)
```

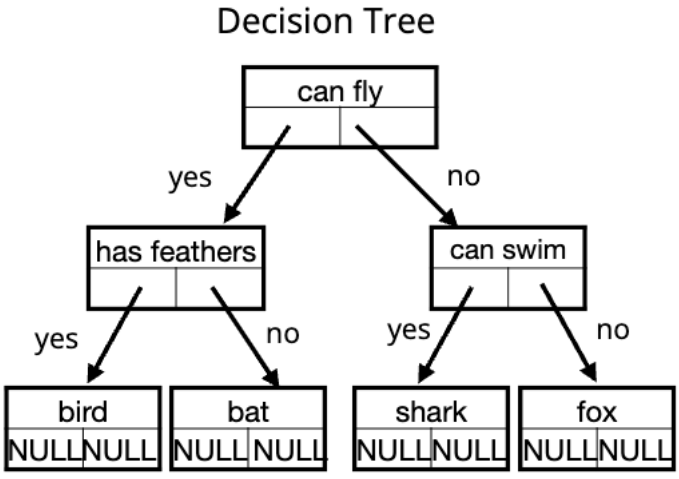
11 Trees

Trees are one of the most fundamental concepts in programming.

Trees are a linked data structure, with nodes and pointers, are are used to represent hierarchical data, organize info for searching, make decisions, etc.

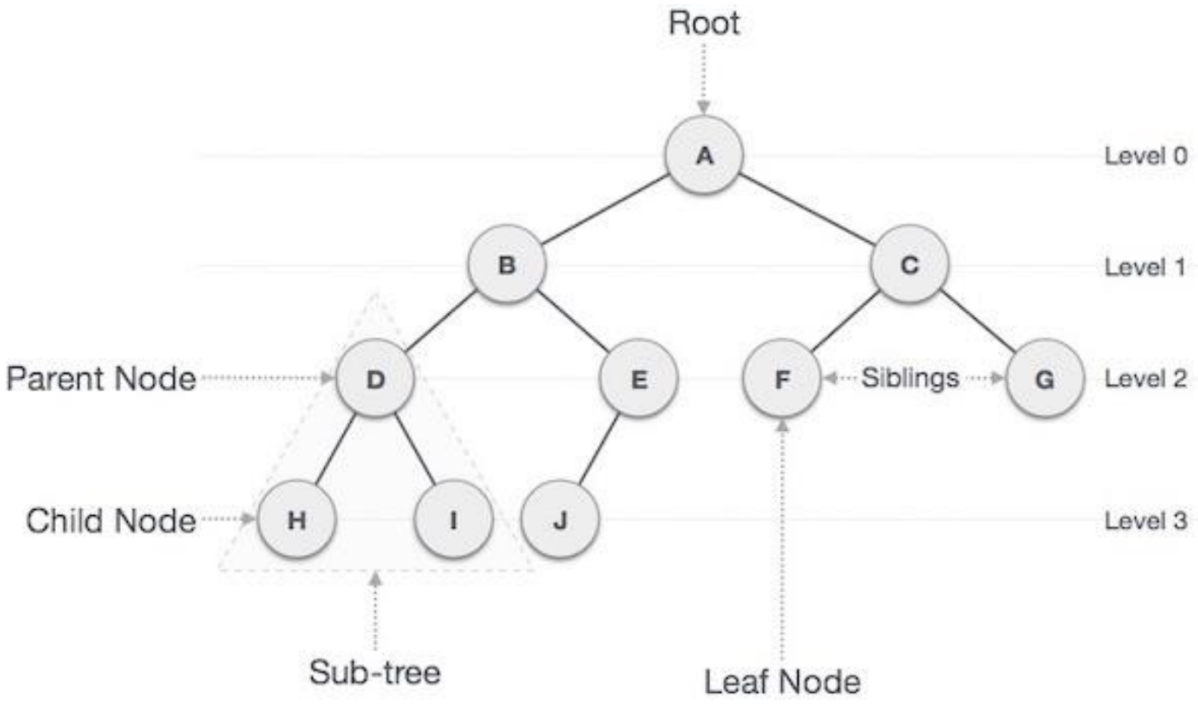
Examples include family trees, search based on car brand -> model -> year -> colour





Trees contain nodes, where every node may have zero or more children. Children nodes are accessed via their parent node, pointing downwards

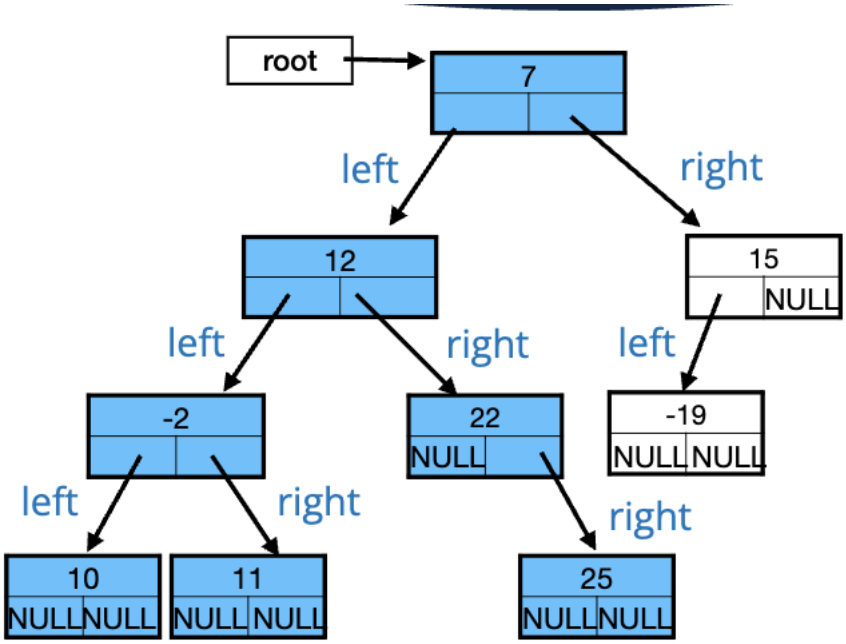
A **binary** tree has 0, 1 or 2 children. Every tree has a **root**, which is the top node.



Implementation

```
typedef struct node {
    int data;
    struct node * left;
    struct node * right;
} node;

new* newNode(int data) {
    node *nptr = (node*) malloc (sizeof(struct node));
    nptr->data = data;
    nptr->left = NULL;
    nptr->right = NULL;
    return nptr;
}
```



11.1 Terminology and Types of Trees

Edges are the relationship between a parent and a child in a tree, drawn as a line and implemented as a pointer. e.g. $\{AB, AC, BH, etc.\}$

A **path** starts from one node and ends at another node, strictly downwards and identifies all nodes and edges along the way. e.g. Path $= ABH$. There is exactly one path from root to any given node.

The **height** of a node is the number of edges in the longest downward path from that node and a leaf node (root down)

Maximum nodes $= 2^{h+1} - 1$

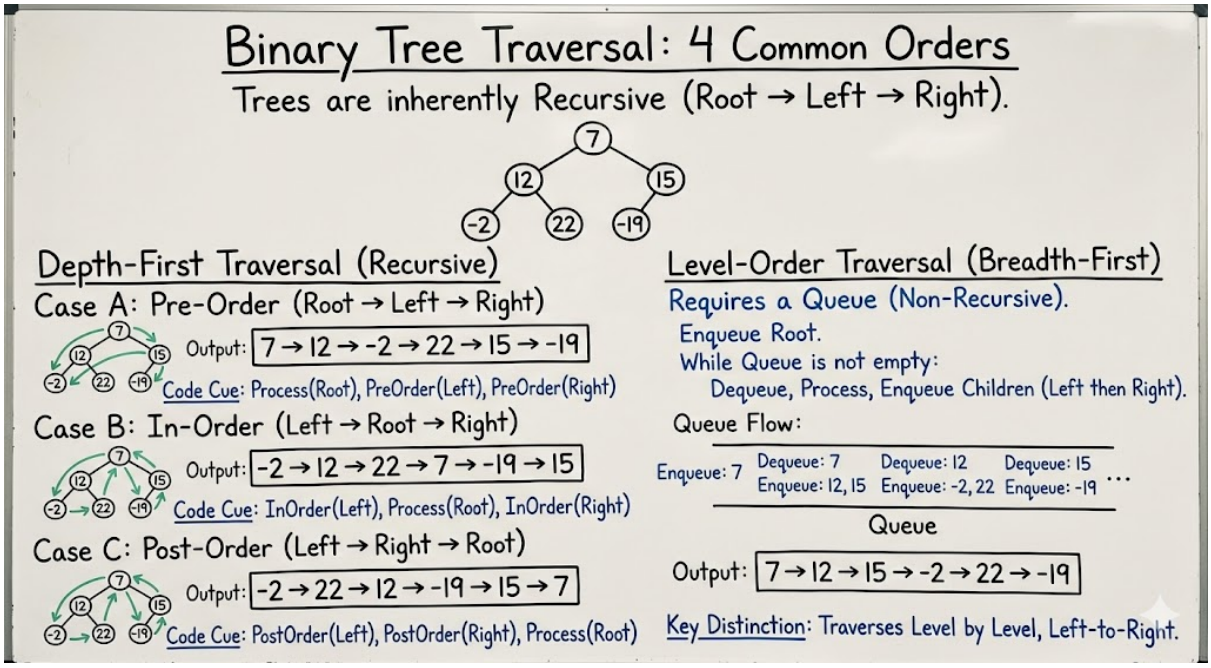
The **depth** of a node is the number of edges from the root to the node, essentially the opposite from the height (leaf up)

The **level** of a node is the depth of the node $+1$

A **subtree** is a given node in the tree and *all* of the nodes below it. Leaf nodes have empty subtrees.

Binary trees are trees with at most two children for any node. A binary tree is **full** if all nodes have exactly zero (i.e. leaf) or two. A binary tree is **complete** if all levels except the last are filled (all nodes in the last level must be as far left as possible).

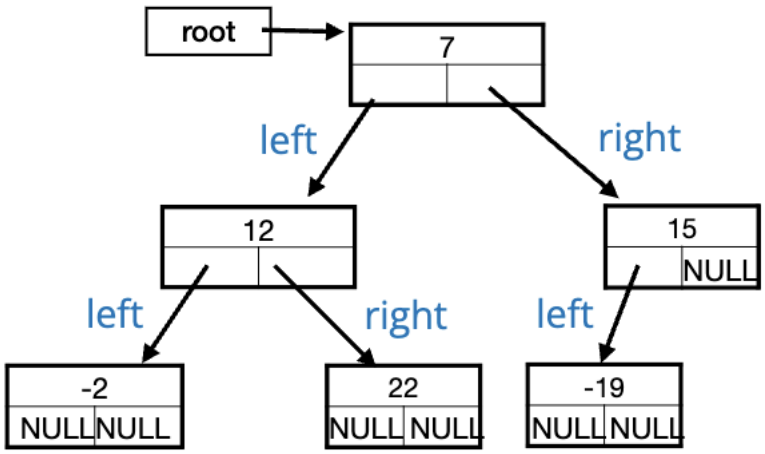
12 Binary Tree Traversal



We can traverse all the nodes, search, add, add at a specific location, delete, delete the entire tree

We can visit all the nodes in a tree starting at the root, but in four common orders: - pre-order - in-order - post-order - level-order

It is important to note that trees, by nature, are recursive.



Example tree:

Pre-order Traversal: Print the root first, then left, then right

1. process the current node
2. traverse the current node's left subtree
3. traverse the current node's right subtree

```
preorderPrint(BinaryTreeNode *cur){
    if (cur == NULL) return; // check
    else {
        printf("%d \n", cur->data);
        preorderPrint(cur->left); // recursion for left
        preorderPrint(cur->right); // recursion for right
    }
}

preorderPrint(root);
```

Output: 7 12 -2 22 15 -19

In-order traversal: Print left child first, then root, then right child

1. traverse the current node's left subtree
2. process the current node
3. traverse the current node's right subtree

```
inorderPrint(BinaryTreeNode *cur){
    if (cur == NULL) return; // check
    else {
        inorderPrint(cur->left); // recursion for left
        printf("%d \n", cur->data);
        inorderPrint(cur->right); // recursion for right
    }
}

inorderPrint(root);
```

Output: -2 12 22 7 -19 15

Post-order traversal: Print value of root last

1. traverse the current node's left subtree
2. traverse the current node's right subtree
3. process the current node

```
postorderPrint(BinaryTreeNode *cur){
    if (cur == NULL) return; // check
    else {
        postorderPrint(cur->left); // recursion for left
        inorderPrint(cur->right); // recursion for right
        printf("%d \n", cur->data);
    }
}

postorderPrint(root);
```

Output: -2 22 12 -19 15 7

Level-order traversal: start at the root, traverse each level from left to right difficult to do recursively, use a queue instead

1. add the root node to a queue
2. while the queue is not empty
    1. remove the first element form the queue
    2. process the node
    3. add any children from left to right to the queue

```
void levelOrderTraversal(BinaryTreeNode *node){
    if (node == NULL) return;
    enqueue(node);
    while(!isEmpty()){
        BinaryTreeNode *temp;
        dequeue(&temp);
        printf("%d,", temp->data);
        if (temp->left != NULL){
            enqueue(temp->left);}
        if (temp->right != NULL){
            enqueue(temp->right);}
    }
}

levelOrderTraversal(root);
```

Output: 7, 12, 15, -2, 22, -19

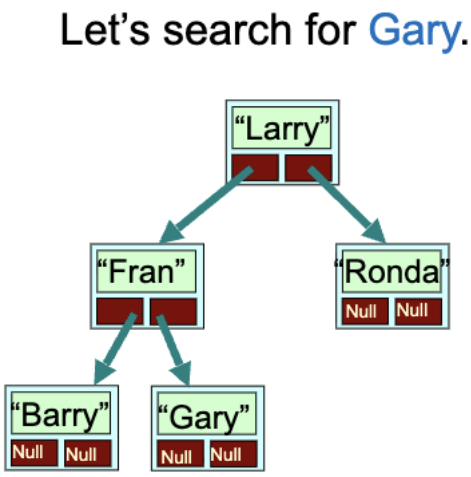
13 Binary Search Trees

Binary Search Trees are an extremely efficient way to store/search for data

A BST is where, for every node X in the tree:

- All nodes in X's left sub-tree must be less than X
- All nodes in X's right subtree must be greater than X

13.1 Searching a BST



Let V be the value to search for, and the outputs either TRUE if found or FALSE otherwise

Iteratively:

```
// declare function
(while ptr!=NULL){
    if (V==ptr->data)
        return 1;
    else if (V < ptr->data)
        ptr = ptr->left; //search the left subtree
    else
        ptr = ptr->right; //search the right subtree
}
return 0 // not found
```

Recursively:

```c
// declare function
while (ptr != NULL){
    if (V == ptr->data)
        return 1;
    else if (V < ptr->data)
        return search(ptr->left, V); // search left subtree
    else
        return search(ptr->right, V); // search right subtree
}
```

The recursive version is more elegant, but less efficient than iterative because this way calls functions must less often (which takes longer than iteration of a loop)

Big O of BST search

When we search a tree, we eliminate 50% of the tree at every step, therefore there are $N \cdot \frac{1}{2^{\text{steps}}} = 1 \Rightarrow 2^{\text{steps}} = N \Rightarrow \text{steps} = \log_2 N$

13.2 Inserting a new value into a BST

To insert a new node, we must place is so that the resulting tree is still a valid BST

Pseudocode:

```
If the tree is empty
    Allocate a new node and put V into it, set the new node as root

Start at the root of the tree
While not done:
    If V is equal to current node, done

    If V is less than current node
        if there is a left child, go left
        else, allocate a new node and put V into it and set
        new node as the left child of the current node

    If V is greater than current node's value
        if there is a right child, then go right
        else, allocate a new node and put V into it, set
        new node as the right child of the current node
```

Iterative:

```
void Insert(BinaryTreeNode **node, int V){
    struct BinaryTreeNode *temp = newNode(V);
    struct BinaryTreeNode *current;
    if (*node==NULL)
        *node = temp;
    else
        current = *node;
        while (current != NULL){
            if (V > (current->data)) // go right
                if (current->right == NULL) // insert if empty
                    current->right = temp;
                else
                    current = current->right;
            else if (V < (current->data))
                if (current->left != NULL)
                    current->left = temp;
                else
                    current = current->left;
        }
}
```

Recursive:

```
void Insert(BinaryTreeNode **node, int V){
    if (*node==NULL)
        BinaryTreeNode *temp = newNode(V);
        *node = temp;
    if (V > ((*node)->data))
        Insert(&(*(node)->right), V); // go right and insert
    else if (V < (node->data))
        Insert(&(*(node)->left), V); // go left and insert
}
```

Big O of BST Insertion is $O(\log_2 n)$

13.3 Deleting a Node from a BST

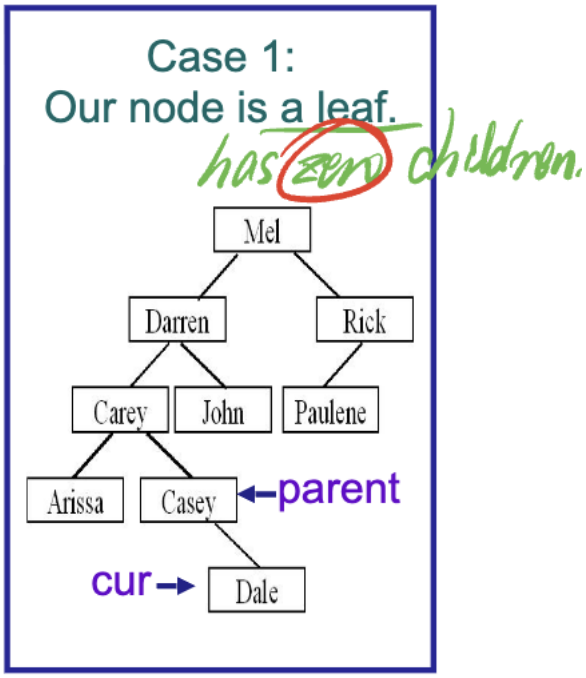
When deleting a node from a BST, we:

1. Find the value V in the tree, with a modified BST search with two variables: a `curr` pointer and a `parent` pointer
2. If the node was found, delete it from the tree, then there are three cases

Step 1: searching for value V

```
BinaryTreeNode *Delete(BinaryTreeNode *node, int data){
    BinaryTreeNode *temp;
    if (node==NULL)
        return NULL;
    else if (data < node ->data)
        node->left = Delete(node->left, data)
    else if (data > node->data)
        node->right = Delete(node->right, data)
}
```

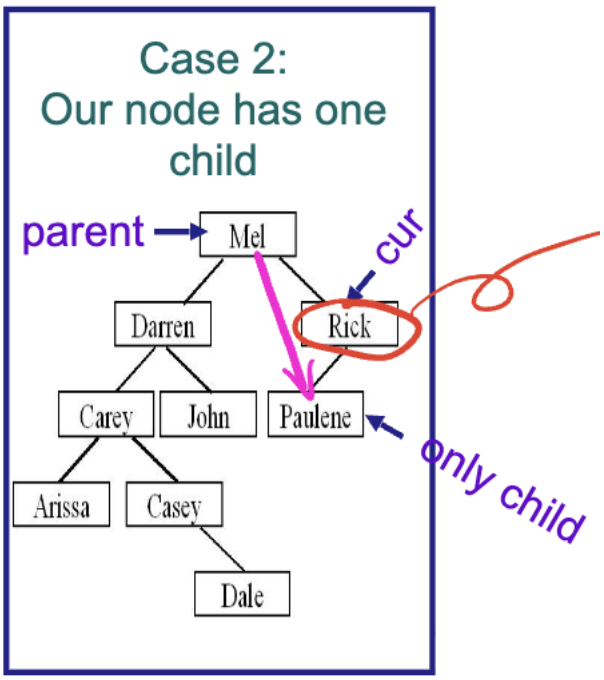
13.3.1 Case 1: Node is a leaf



If we want to delete curr, which has zero children (i.e. a leaf node), we have two subcases:

1. Unlink the parent node from the curr node by setting the parent's appropriate link to NULL
2. If the curr node **is the root node**, set the root value to NULL

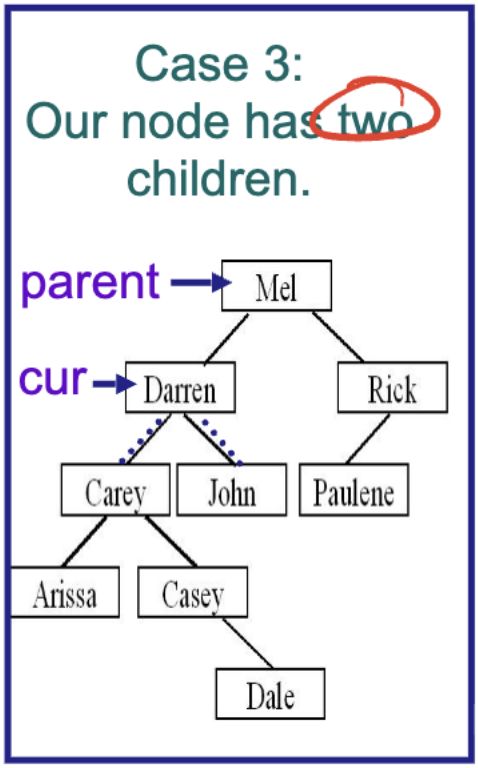
13.3.2 Case 2: Target Node has One Child



If we want to delete curr, which has one child, we have two sub cases:

1. Relink the parent node to curr's only child
2. If the curr node **is the root node**, set the curr node's only child as the root

13.3.3 Case 3: Target Node has Two Children



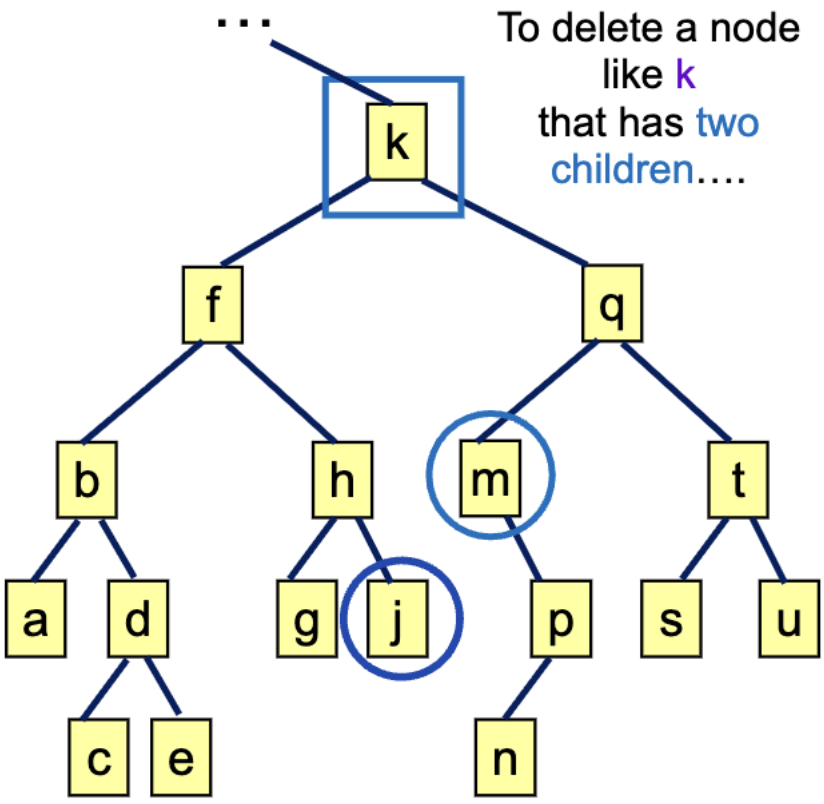
We need to find a replacement for our target node that still leaves the BST consistent

To delete a node K that has two children, we either replace its value with:

1. K's left subtree's largest-valued child (case 1) or
2. K's right subtree's smallest-valued child (case 2)

Because a BST satisfies: (left subtree) $<$ root $<$ (right subtree)

Also, by definition, the left subtree's maximum value can't have a right child, and the right subtree's minimum value can't have a left child



Implementation:

```
BinaryTreeNode * Delete(BinaryTreeNode *node, int data){
    BinaryTreeNode *temp;
    if(node==NULL)
        return NULL;
    else if(data < node->data)
        node->left = Delete(node->left, data);
    else if(data > node->data)
        node->right = Delete(node->right, data);
    else{ /* This is the case when data=node->data. */
        /* Now We can delete this node and replace with either minimum element
           in the right sub tree or maximum element in the left subtree */
        if(node->right && node->left){
            /* Here we will replace with minimum element in the right sub tree */
            temp = FindMin(node->right);
            node -> data = temp->data;
            /* As we replaced it with some other node, we have to delete that node */
            node -> right = Delete(node->right, temp->data);
        }
        else {
            /* If there is only one or zero children then we can directly
               remove it from the tree and connect its parent to its child */
            temp = node;
            if(node->left == NULL)
                node = node->right;
            else if(node->right == NULL)
                    node = node->left;
            free(temp); /* temp is no longer required */
            }
        }
    return node;
}
```

Step 1

Case 3

Case 1&2

13.4 Finding Min & Max of a BST

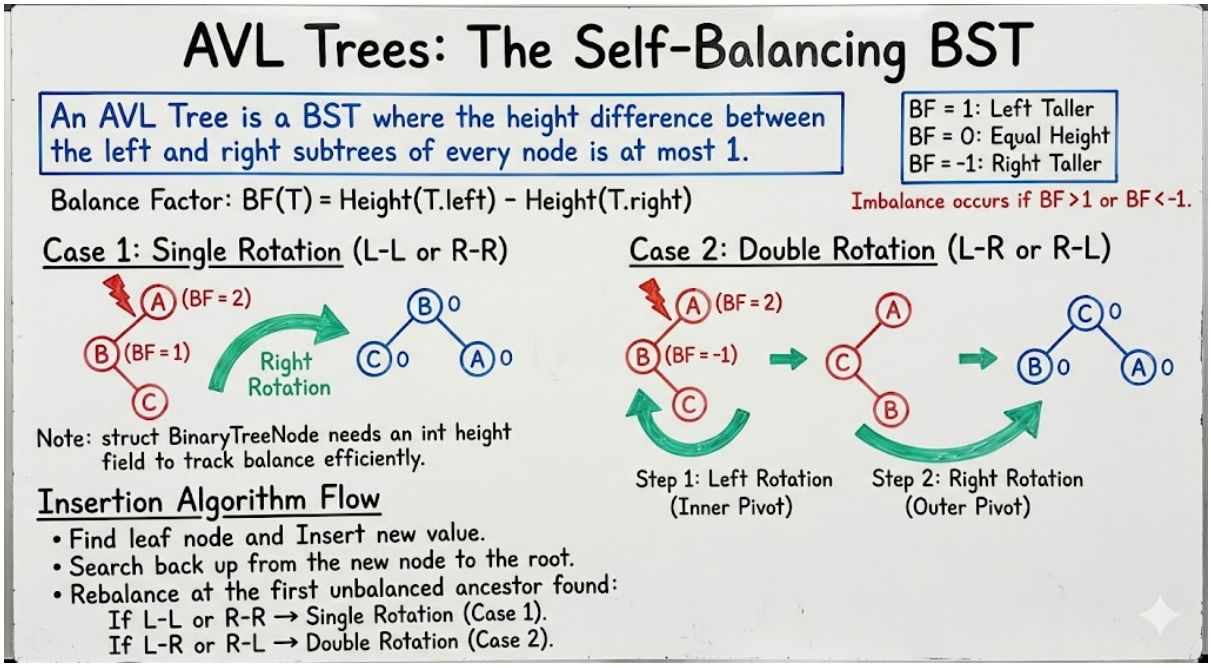
The minimum value is located at the left-most node

```
// declare
if (node == NULL)
    return NULL;
else if(node->left != NULL)
    return FindMin(node->left);
else
    return node;
```

The maximum value is located at the right-most node

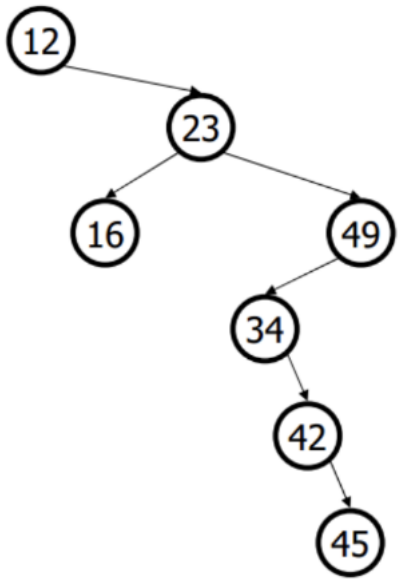
```
// declare
if (node == NULL)
    return NULL;
else if (node->right != NULL)
    return FindMax(node->right);
else
    return node;
```

14 AVL Trees and Rotations



When we inserted 49, 34, 42, and 45 into the BST 12, 23, and 16, we are left with an unbalanced tree

A **balanced search tree** is a BST in which the height of the left and right subtree of *any node* differ by not more than 1. In this case, the right subtree has a height of 3



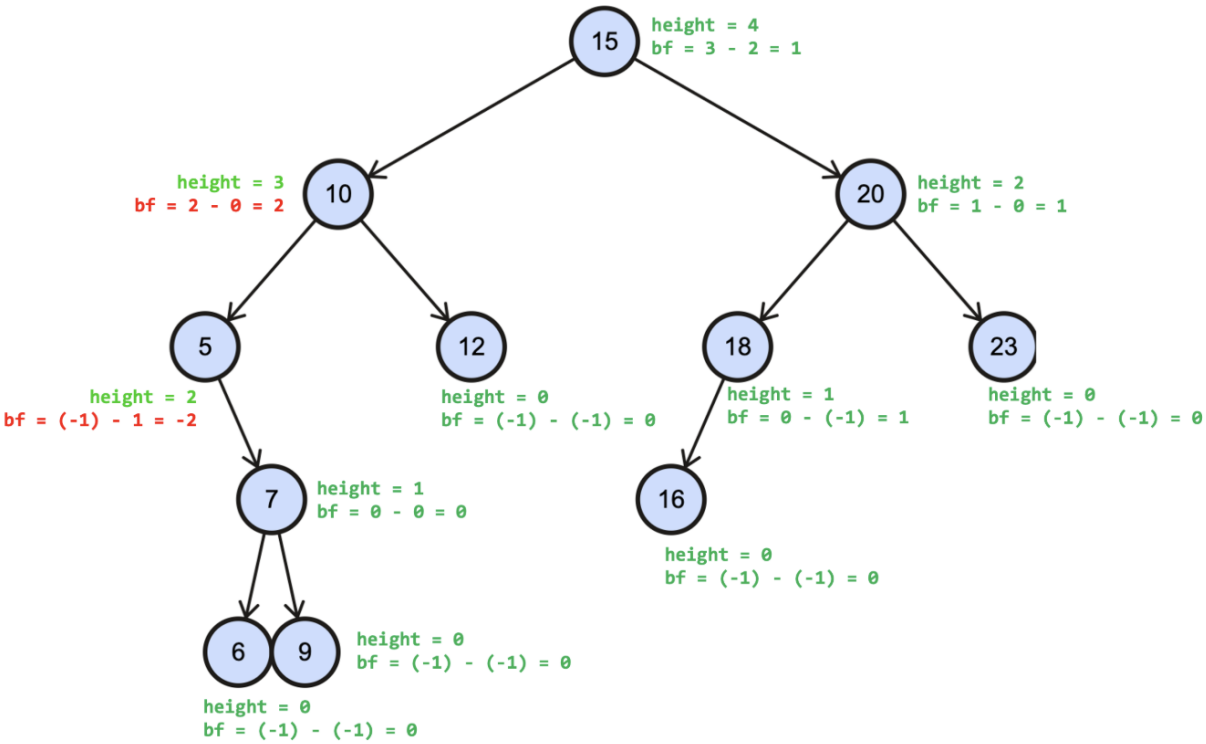
An **AVL tree** is a self-balancing binary search tree in which for every node, the heigh difference between its left and right subtrees is **at most 1**

$$height(n) = \begin{cases} -1 & n = \text{null} \\ 0 & n = \text{leaf} \\ 1 + \max(\text{height of children}) & n \neq \text{leaf} \end{cases}$$

i.e., a BST is balanced when, for any node, its balanced factor is 1, 0, or -1

Therefore: **Balance Factor:** $BF(T) = Height(T.left) - Height(T.right)$

$$\begin{cases} BF(T) = 1 & \text{Left subtree is one level taller than right subtree} \\ BF(T) = 0 & \text{Left and right subtrees have the same height} \\ BF(T) = -1 & \text{Right subtree is one level taller than left subtree} \end{cases}$$



Implementation:

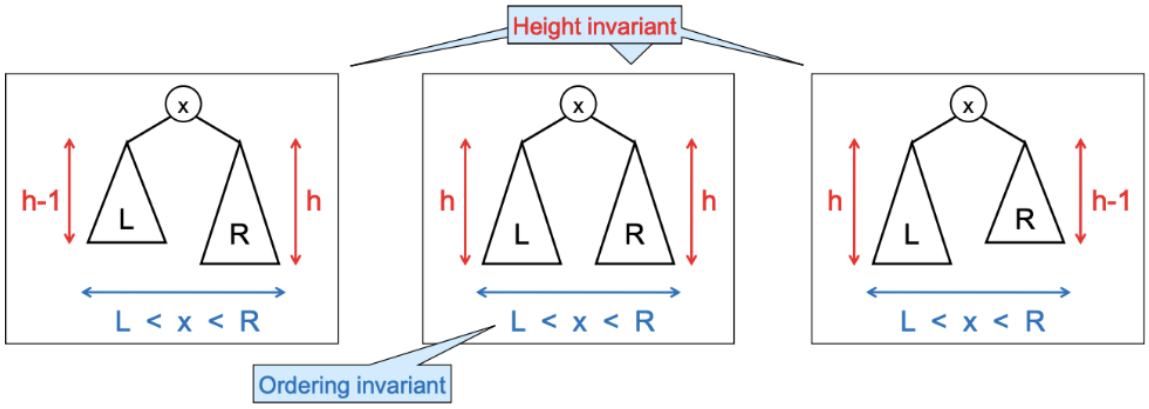
```
int getHeight(struct BinaryTreeNode *t) {
      int heightLeft, heightRight, heightval;
      if (t == null)
          return heightval = -1;
      else{
          heightLeft = getHeight(t ->left);
          heightRight = getHeight(t->right);
          heightval = 1 + ((heightLeft > heightRight)? heightLeft : heightRight);
                      // max(height(X->left), height(X->right))+1
      }
      return heightval;
}

 int getBalance(struct BinaryTreeNode *t) {// Get Balance factor of node t
     if (t == NULL)
       return 0;
     return getHeight(t->right) - getHeight(t->left);
 }
```

To keep track of heights: we can keep track of subtree height as a field in the struct:

```
struct BinaryTreeNode {
    int data;
    int height; // here
    struct node* left;
    struct node* right;
}
```

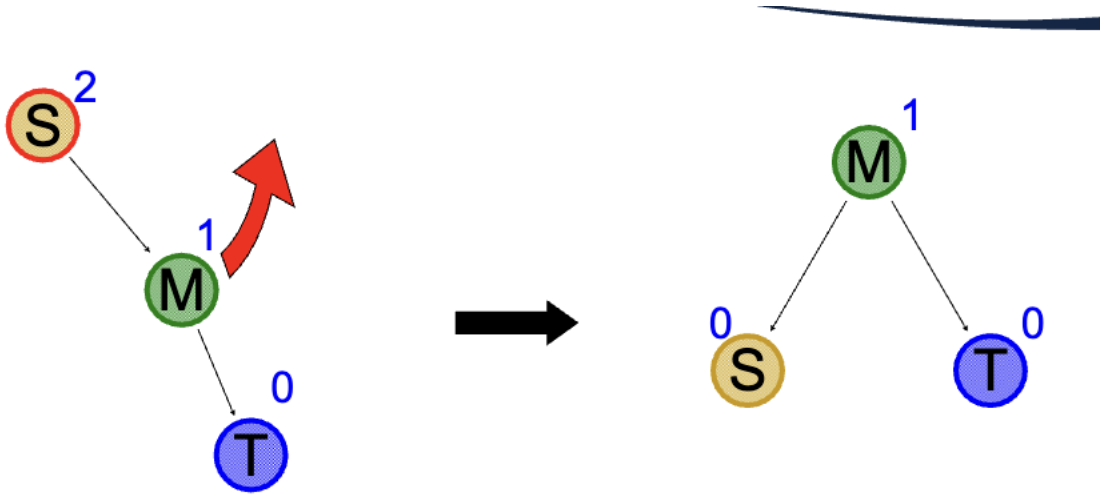
If a node has become out of balanced in a given direction, we can rotate it in the opposite direction



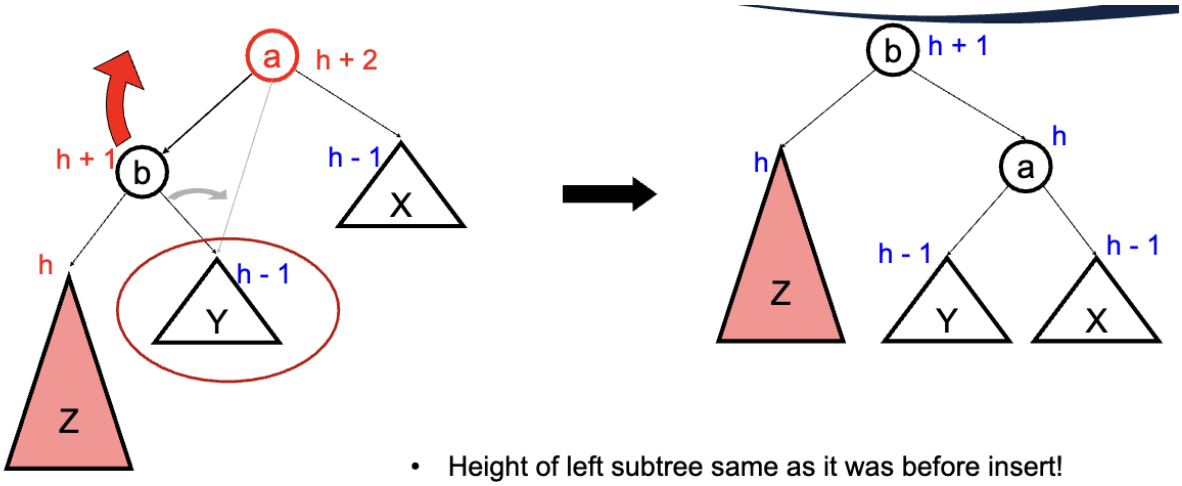
14.1 Case 1: Left-Left or Right-Right

This imbalance can be caused by either inserting into left child's left subtree or right child's right subtree

We can fix this by considering that a right child (green) could legally have its parent (left) as its left child



Below, by rotating and letting b be the root, we see that the BST becomes balanced

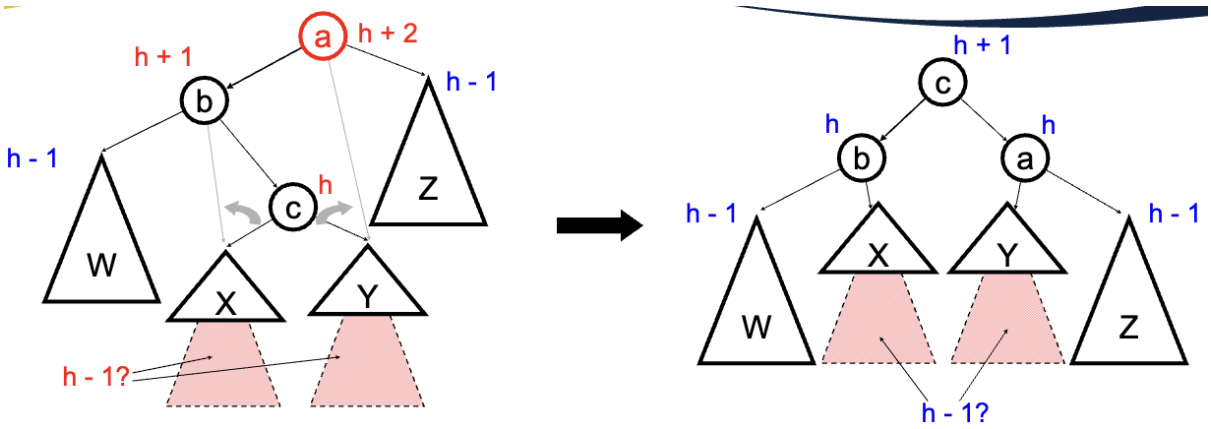


- Height of left subtree same as it was before insert!
- Height of all ancestors unchanged
  - *We can stop here!*

14.2 Case 2: Left-Right or Right-Left

This imbalance can be caused by either inserting into the left child's right subtree or inserting into the right child's left subtree

We can fix this with a double rotation



Initially: insert into either X or Y unbalances tree (root balance goes to 2 or -2)
"Zig zag" to pull up c – restores root height to h+1, left subtree height to h

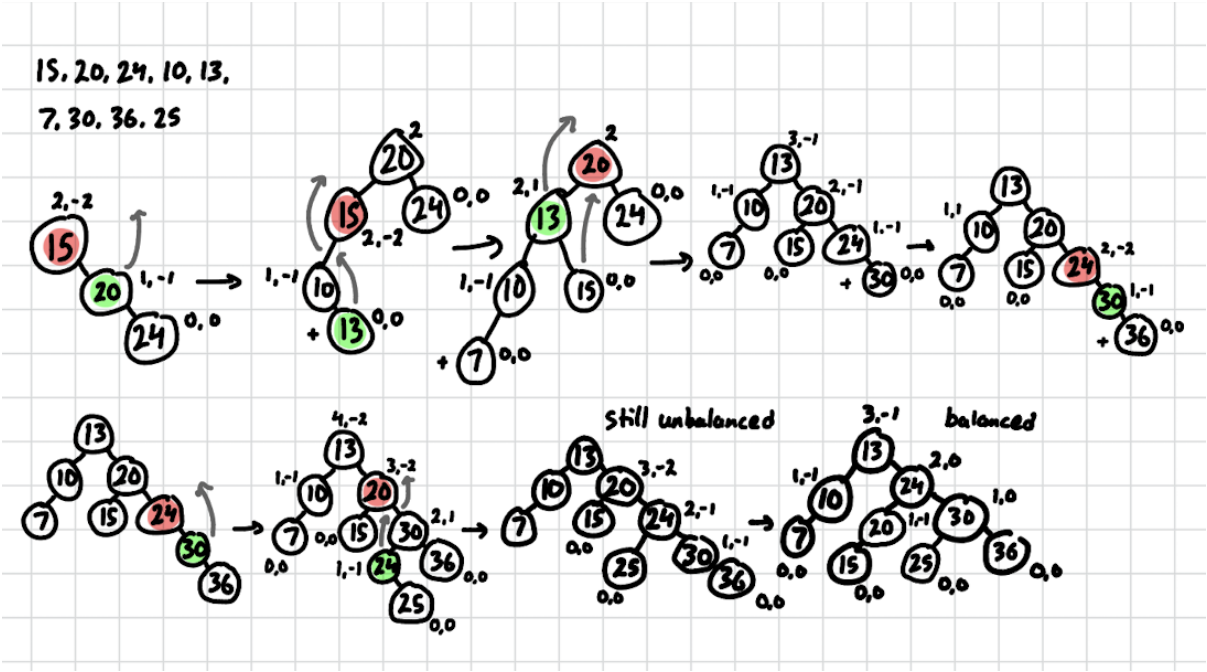
Insert Algorithm

- Find a spot for value
- Insert new node
- Search back up looking for imbalance
  - case 1: single rotation
  - case 2: double rotation



14

Drawn Example:



Code Implementation:

```c
#include <stdio.h>
#include <stdlib.h>

// An AVL tree node
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// A utility function to get the height of the tree
int height(struct Node *N) {
    if (N == NULL)
        return -1;
    return 1 + max(height(N->left), height(N->right));
}

// Helper function that allocates a new node with the given key and
↪  NULL left and right pointers.
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 0; // new node is initially added at leaf
    return node;
}

// A utility function to right rotate subtree rooted with y
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2; // T2 is the subtree put as left child of root

    // Update heights
    y->height = height(y);
    x->height = height(x);

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
```

15 Red Black Trees

Not tested on our final.

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

- Nodes are assigned a colour, red or black
- Starting with black as the root node
- Balancing is determined by its colour and their orientation and not its balance factor

Properties of RB Trees

- every node is either red or black
- the root of the three is always black
- red nodes cannot have red children, two consecutive red nodes on any path is not allowed
- every path from a node to its descendant null nodes has the same number of black nodes
- all leaves/null nodes are black

Balance in RB Trees

- not the same balancing rules for AVLs
- RB trees can either change the colour of a node or rotate the nodes to maintain order
- this depends on the rule being violated
- nodes are always red on insertion (root node becomes black after insertion_
- we consider NULL leaves as nodes in our tree

Cases for Insertion

Rules:

1. insert new node as RED
2. if new node is root, node changes to BLACK
3. if parent of new node is BLACK, no action
4. if parent of new node is RED
    1. if the uncle of new node is red, recolour parent, uncle, and grandparent (single)
    2. if the uncle of the new node is black, rotate and/or recolour

Cases for Deletion

1. find the node to be deleted
2. replace the node with its successor (if two children)
3. if the deleted node or its success is red, just remove it
4. if the deleted node and its successor are black, fix the double black issue:
    1. if the sibling is black and has a red child, rotate and recolour
    2. if the sibling is black and has two black children, recolour and move the problem up the tree

16 Heaps and Priority Queues

A priority queue is an abstract data structure where

- each element has a priority
- elements with higher priority are served before lower-priority ones

Instead of a normal queue being FIFO, a priority queue extracts elements with the highest priority

A common application of a priority queue is **heap**

16.1 Heaps

A heap is a complete binary tree that satisfies the heap property:

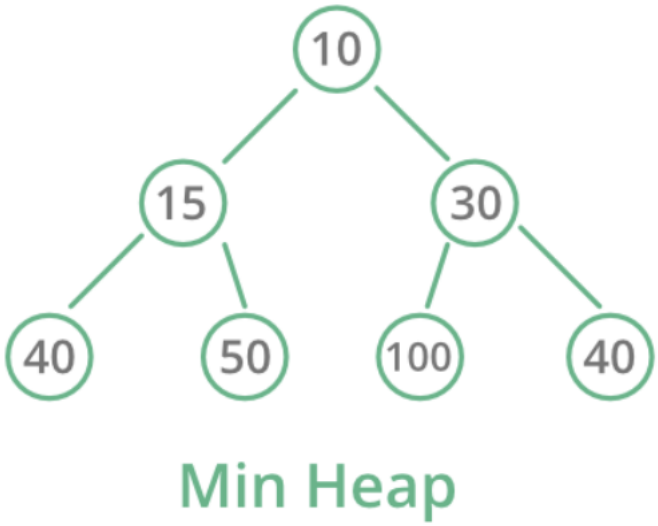
- in a minheap, each parent $\leq$ its children
- in a maxheap, each parent $\geq$ its children

Heaps are not good for searching, but are good for storing lots of sorted data. This is because a heap is designed to support two operations, **insert** and **extract**

If we were to search a heap, it only tells us something about parent-child priority relationships, not ordering between sibling or across different branches

16.1.1 Minheaps

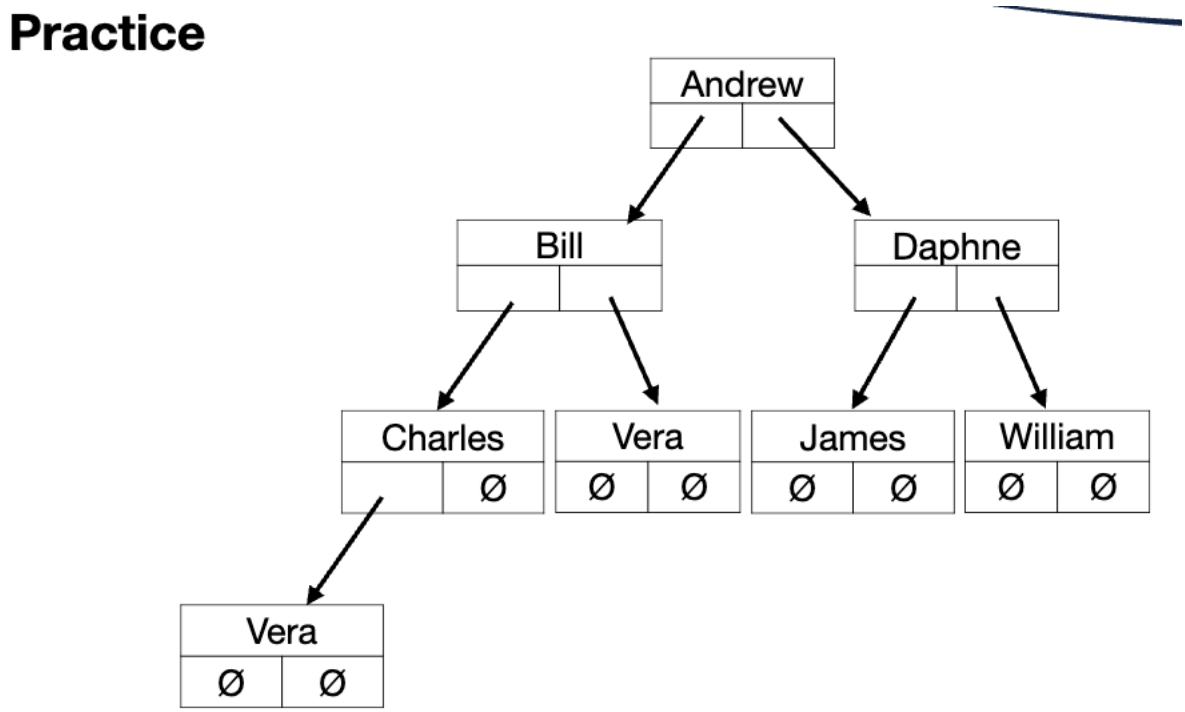
The value at a given node is always smaller than or equal to the values of the node's children



Extracting the smallest value from a minheap

The smallest value is 10. We can copy the value of the far right node of the lowest level (40) to the root node (10), then delete the last node (40)

Then, to satisfy the minheap property, we need to **sift** the root node (now 40)

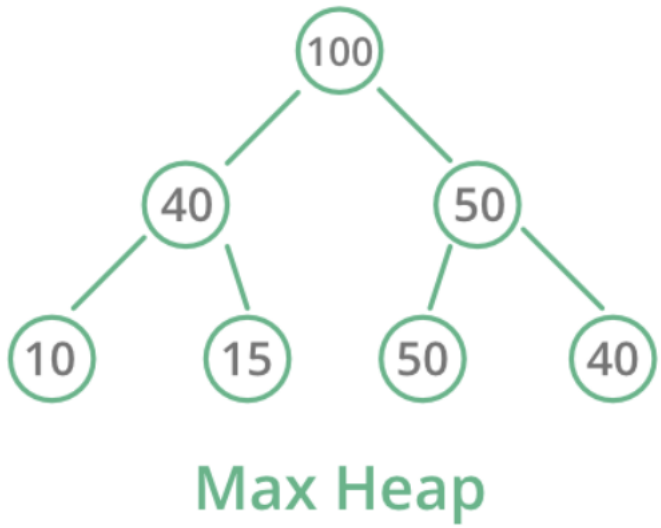


Vera will replace the root node. The Vera node is deleted. Vera is sifted with Bill because Bill is bigger than Daphne. Vera is sifted with Charles because Charles is bigger than Vera (bill's child).

A similar process occurs with maxheap

16.1.2 Maxheap

The value at a given node is always larger than or equal to the values of the node's children

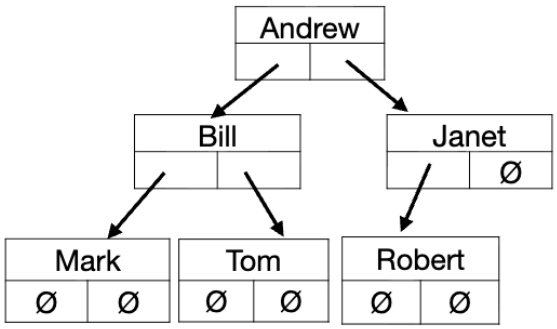


Heap as an array

We sort nodes into an array from biggest to smallest, from root to lowest level

Implementing a Heap as an Array

- Then row 3



0	Andrew
1	Bill
2	Janet
3	Mark
4	Tom
5	Robert
6	

It is easy to notice that the root is at position 0 (andrew), and the bottom rightmost node is at the lowest position (5, Robert).

We can easily find the next open slot to add to as well (position 6)

Furthermore, to find the children of a node, we can implement a function with the parent's index:

`leftChild(parent)=2*parent + 1` `rigthChild(parent)=2*parent + 2`

For example, 2 * 2(Janet) + 1 = 5 (Robert) For example, 2 * 1 (Bill) + 2 = 4 (Tom)

In reverse, to find the parent of a child with the index:

`parent(child)=(child-1) / 2`

For example, (5 (Robert) - 1) / 2 = 2 (Janet) For example, (4 (Tom) - 1) / 2 = 1.5 $\approx$ 1 (Bill) (any fractional part of the result is discarded)

16.1.3 Extract Item from Heap (using array)

We can copy Andrew, then copy `heap[len-1]` (Julia) to `heap[0]`

Once we remove the last node of the heap, we decrease the length -1

Then, compare `leftChild(0)` to `rightChild(0)`. In our case, 1 (Bill) $<$ 2 (Janet)

So we swap Bill with Julia, then compare the new children `leftChild(1)` to `rightChild(2)`. In our case Mark $\not<$ Julia, so don't swap

Inserting a Value

First, add a new node (Charles) to an open slot, then fix any problems that arise

1. Change then length to length + 1
2. Compare new child (Charles) to parent (Janet). In our case, swap
3. Then, compare again. In our case, Charles $>$ Bill, then don't swap

16.2 Stability

The stability problem in heaps refers to the fact that heap-based sorting algorithms are not stable by default. When two elements are equal, their relative positions may change during the swapping operations. Elements with the same value or priority are not extracted in FIFO order (i.e., not stable).

To fix this problem, we may augment elements with levels of urgency.

Example: The Canadian Triage and Acuity Scale has 5 levels:

1. Resuscitation
2. Emergent
3. Urgent
4. Less Urgent
5. Non-Urgent

Big O: The O() of heap operations is $O(\log_2 n)$, because:

Heaps are complete binary trees, i.e. a balanced tree, we know that its height is:

$$\text{nodes} = 2^{h+1} - 1$$

$$n + 1 \leq 2^{h+1} \Rightarrow \log(n+1) \leq h + 1$$

$$h \geq \log_2(n+1) - 1 \Rightarrow h = O(\log_2 n)$$

17 Hash Tables

Definition: Hash tables are often the most efficient way to search for data. It is a data structure that implements a map (also called associative array) abstract data type. An ADT consists of

- A collection of unique keys (values to find an entry quickly)
- A collection of values, where each key is associated with one value or set of values

It acts similarly to a dictionary, so when given a key, we get its corresponding value.

17.1 Map Operations

`get(key k)`

- returns the value associated with the key k
- returns NULL if key k is not in the map

`put(key k, value v)`

- if the map does not contain an entry with key k, then insert the new entry (k, v)
- if the map does contain an entry with key k, then replace the old value with the new value v

`Remove(key k)`

- If the map does not contain an entry with key k, then do nothing. Otherwise, remove the entry.

`Size()`

`isEmpty()`

17.2 Hash Function and Collisions

We write a `hashFunc` that takes a key and converts it into an index in the array, between 0 and `array_size - 1`. The hash function determines where each key should go in the array (main storage).

$$h(x) = x \ \% \ size$$

```
hashFunc(idNum):

ARRAY_SIZE = 100000;
int bucket = idNum & ARRAY_SIZE;
return bucket;
```

The process of taking a key and "chopping it up and mixing it" into an index is why we call is "hash".

A collision is a condition where two or more values both "hash" to the same bucket in the array.

There are many schemes for dealing with collisions, two most popular ones are **closed hash table** (= open addressing) and **open hash table** (= separate chaining)

17.3 Closed Hash Table

In this method, when there is a collision, we store the colliding key and value to the next index.

Therefore, for **linear probing**, the hash function becomes:

$$\text{index } = (k + i) \ \% \text{ size}$$

where $f(i) = i$

Deleting in linear probing: If we delete an element by setting its slot to `NULL`, we break the probe chain. Instead, we insert a deleted flag in a certain index, so that if we want to delete a value with the same index but shifted due to linear probing, we know to delete from the next index according to the flag

Similarly, for **quadratic probing,** the hash function is there same, but $f(i) = i^2$

$$\text{index } = (k + c_1 i + c_2 i^2) \ \% \text{ size}$$

This method is fast, has a fixed table, but usually requires a large hash table to avoid collisions

Similarly, for **double hashing**, we are given two functions. **Example:** $h_1(k) = k$ and $h_2(k) = 1 + (k\%(m-1))$

$$\text{index } = (h_1(k) + ih_2(k)) \ \% \text{ size} \Rightarrow \text{index } = (k + i(1 + k\%10)) \ \% \ 11$$

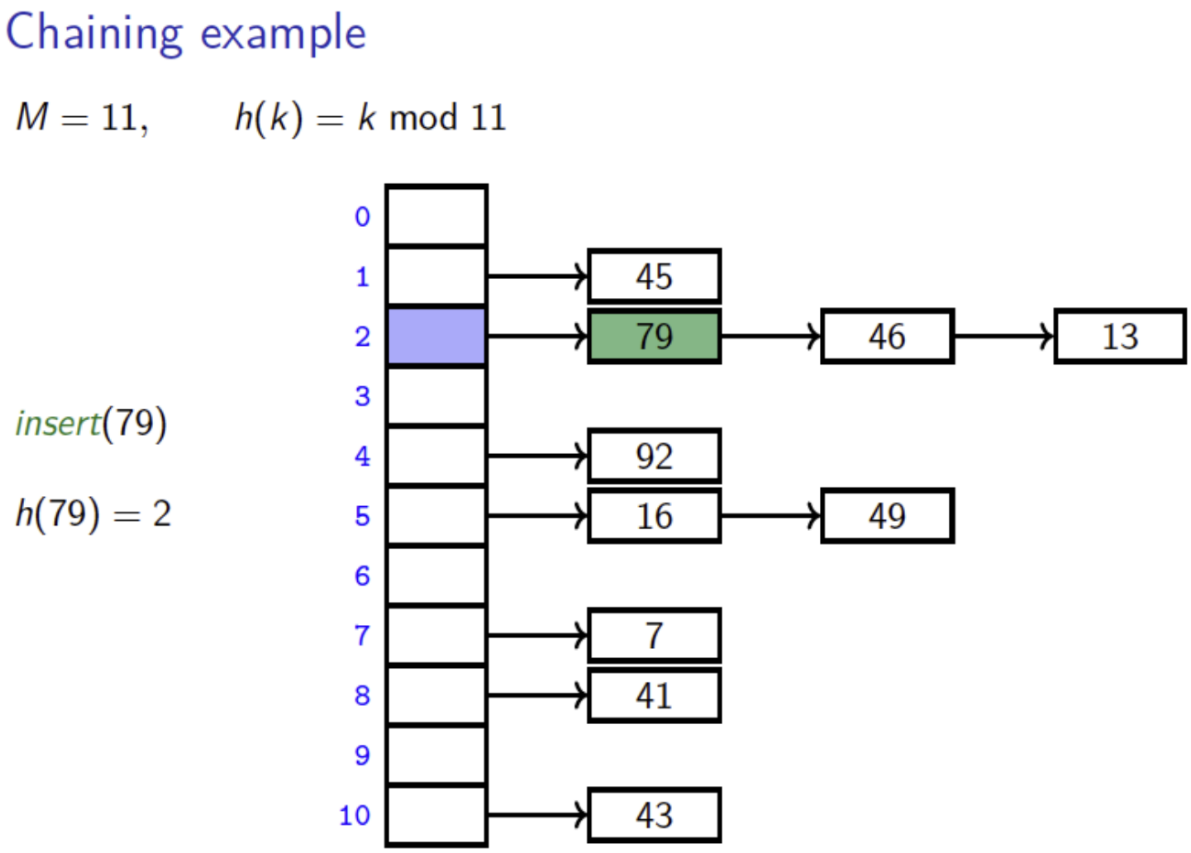
Closed Hash Table Efficiency

Case 1: An unsuccessful search/insert in an open address hash table with load $\alpha = \frac{n}{m} \leq 1$ is at most: $\frac{1}{1-\alpha}$ under simple uniform assumption

Case 2: A successful search in an open address hash table with load α is at most: $\frac{1}{a} \ln \frac{1}{1-\alpha}$ under simple uniform assumption

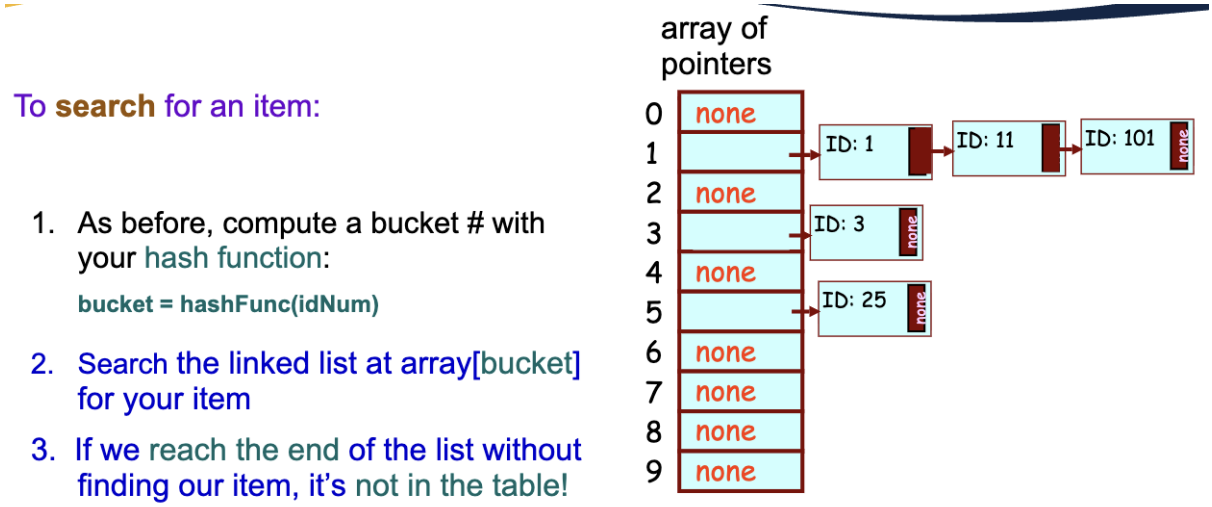
17.4 Open Hash Table

In this method, when we map any key with the hash function, we insert the value in a chain, i.e. a linked list, at that index. We could also use a different data structure, like an AVL tree



Example: Insert $1, 3, 11, 25, 101$

1. Compute a bucket # with the has function
2. Add your new value to the linked list at `array[bucket]`



If you plan to repeatedly insert and delete values into the hash table, then the Open table should be used. It may require less memory, and it is easy to delete values. Although, it may be slower because we need dynamic memory and an extra data structure

Open Hash Table Efficiency

The worst case for a hash table is $O(n)$ for search and $O(n)$ for insert. However, in

practice, hash tables work really well and the worse cast almost never happens.

Simple Uniform Hashing: Every k is equally likely to hash to any of the m buckets

We define the current load to be:

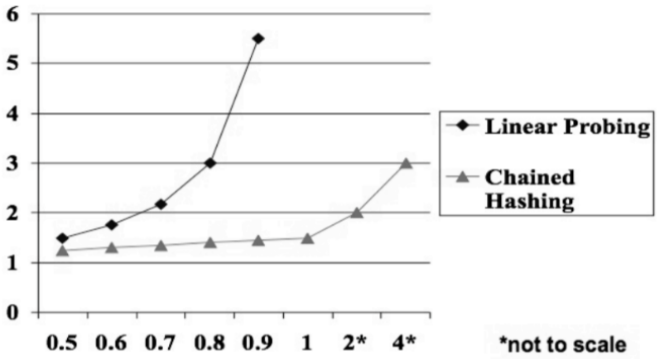
$$\alpha = \frac{\text{N items distrbuted}}{\text{over M buckets}} = \frac{N}{M} = \text{ number of items per bucket}$$

Case 1: An unsuccessful search takes $\theta(1+\alpha)$ time, by $O(1)+\alpha$, because it hashes once, then scans the whole bucket

Case 2: A successful search takes $\theta\left(1+\frac{\alpha}{2}\right)$ time on average. Since there are on average $\frac{n-1}{m}$ keys in that list besides the target key, on average half of them will be searched before finding the target:

$$S_\alpha = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}$$

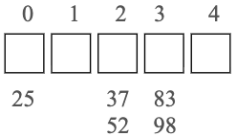
Average number of searches during a successful search as a function of the load factor α



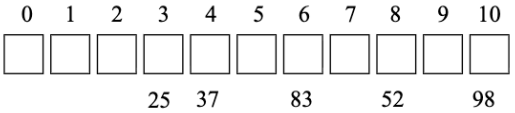
Idea: Create a larger table and then rehash all the elements into the new table.

Open Hash Table:

$h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$



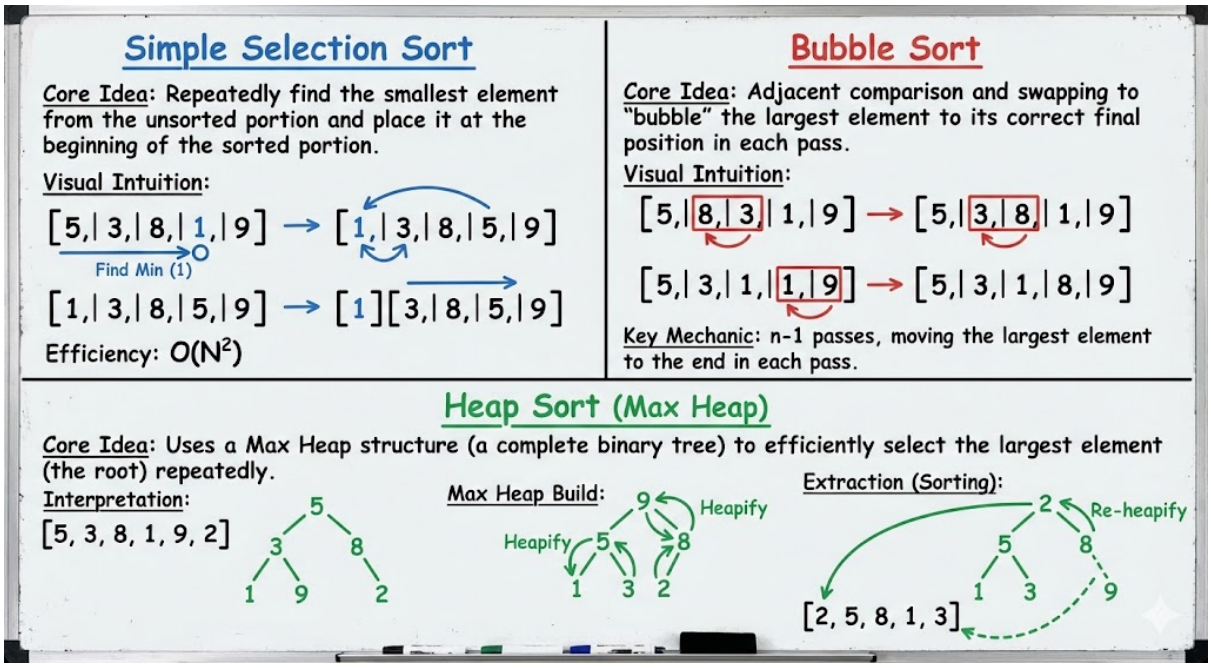
Attention! Positions of original keys will change!



18 Selection Sorting

Sorting deals with a sequence of elements in a list and use some rearrangement strategy to order the elements with the goal to get items of data in the same order as the index of the storage.

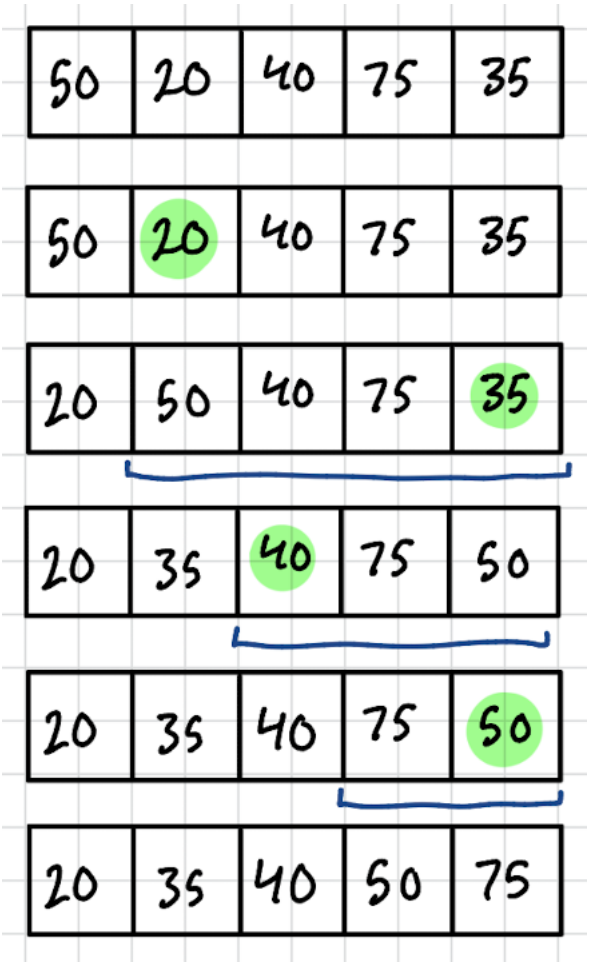
We sort for faster searching, verifying duplicates, etc.



18.1 Simple Selection Sort

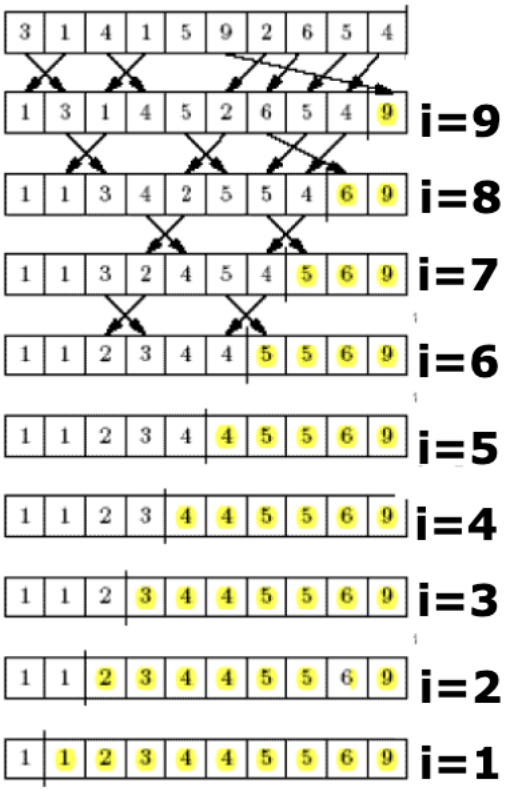
Average case: $O(N^2)$

1. Start at the beginning of the array and find the smallest element
2. Swap it with the first element, then search the remaining unordered elements for the smallest
3. Swap it with the second element, and continue until the array is fully sorted



18.2 Bubble Sort

Bubble sort makes $n-1$ passes through the data, where elements are "bubbled" into place by comparing with adjacent elements. The average number of swap is $\frac{n(n-1)}{4}$




```
void bubbleSort (int arr[]){
    int n = sizeof(arr)/sizeof(arr[0]);
    for (int i = n-1; i>0; --i){
        for (int j = 0; j<i; ++j){
            if (arry[j]>arr[j+1]){
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}

int swap(*x,*y) {
    int temp= *x;
    *x = *y;
    *y = temp;
}
```

18.3 Heap Sort

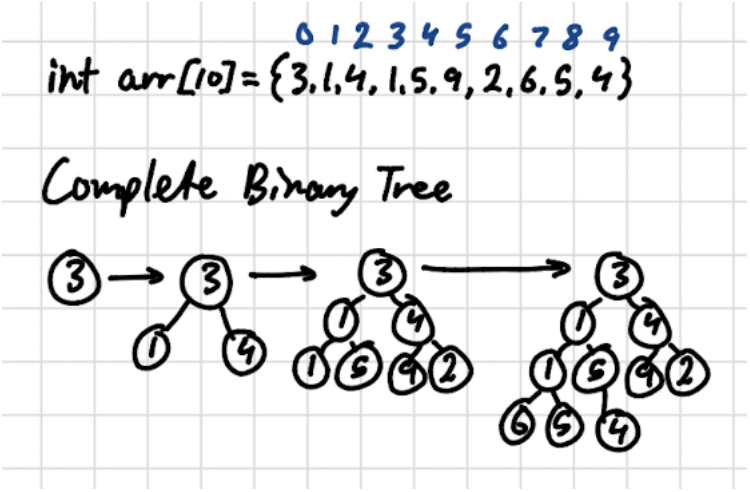
Heap sort works by first building a max heap, then repeatedly removing the root to form the sorted sequence

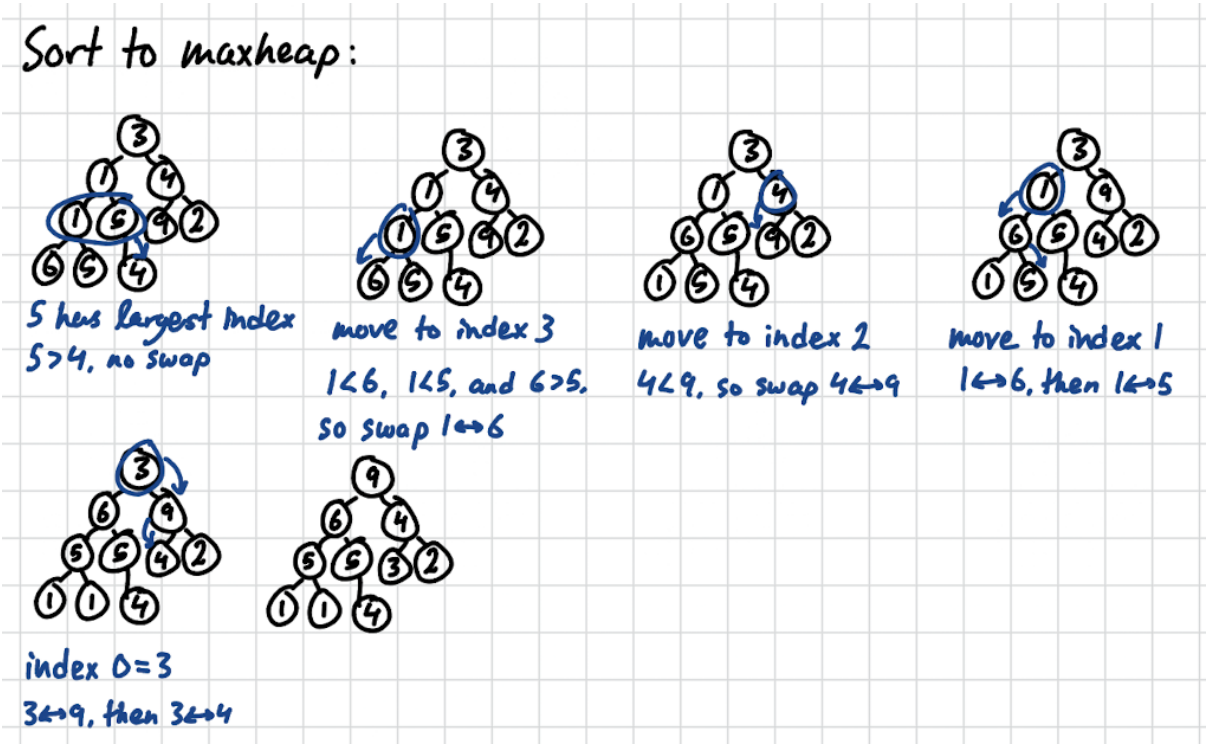
1. Interpret array as complete binary tree
2. Build the max heap:
    1. find the index of the last non-leaf node $\left[\left(\frac{n}{2}-1\right)\right]$
    2. start from this node and moving backward to the root, do the following:

1. compare the parent with its children
2. if the largest child is greater than the parent, swap them
3. continue this process (heapify) until all non-leaf nodes have been visited

3. Extract elements to sort the array:
   1. swap the root with the last element in the heap
   2. reduce the heap size by one
   3. re-heapify the reduced heap
   4. repeat until all elements have been extracted

Example: `int arr[10]={3,1,4,1,5,9,2,6,5,4}`





Sort Array:

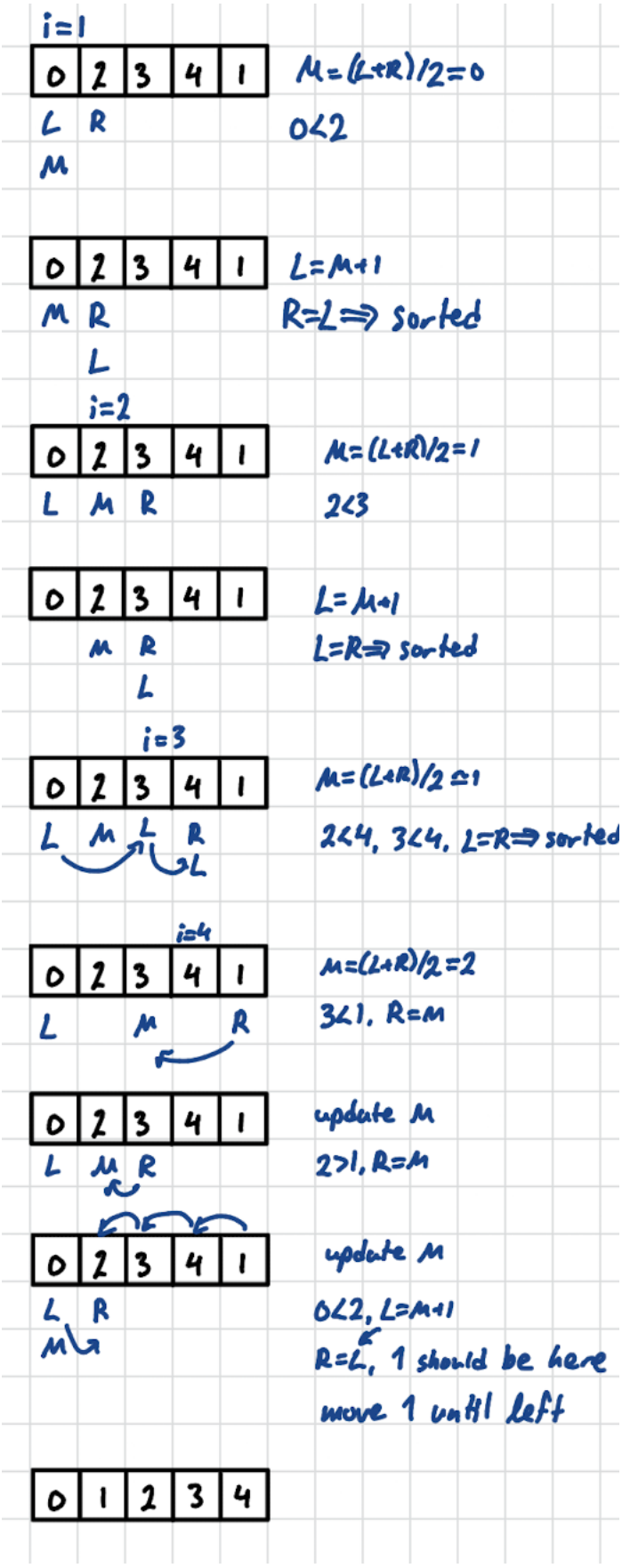
Maxheap array:



19 Binary Search

We use binary search to find the correct position for each element. We often use `mid = (left + right)/2` as the first step

1. Locate middle element in the sorted sequence
2. Compare the middle element with the target element to be inserted
    1. Case 1: If the middle element is equal to the target element, done
    2. Case 2: If the new element is larger than the middle element, sort the right side of the sorted sequence
    3. Case 3: If the new element is less than the middle element, sort the left side of the sorted sequence


i=1
0 2 3 4 1
L R
M
M=(L+R)/2=0
0<2
0 2 3 4 1
M R
L
L=M+1
R=L⇒ sorted
i=2
0 2 3 4 1
L M R
M=(L+R)/2=1
2<3
0 2 3 4 1
M R
L
L=M+1
L=R⇒ sorted
i=3
0 2 3 4 1
L M L R
M=(L+R)/2=1
2<4, 3<4, L=R⇒ sorted
i=4
0 2 3 4 1
L M R
M=(L+R)/2=2
3<1, R=M
0 2 3 4 1
L M R
update M
2>1, R=M
0 2 3 4 1
L R
M
update M
0<2, L=M+1
R=L, 1 should be here
move 1 until left
0 1 2 3 4

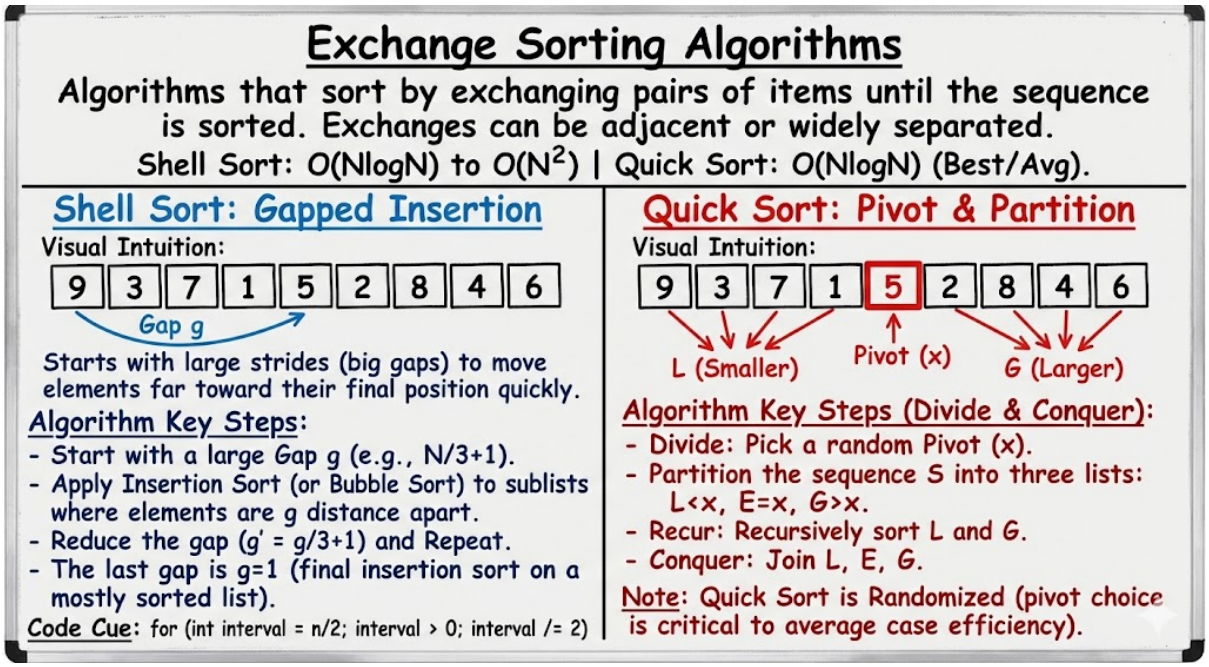
```
void sort(int arr[]){
    int n = sizeof(arr)/sizeof(arr[0]);
    for (int i=1; i<n; ++i){
        int tmp=arr[i];
        int left = 0;
        int right = i;
        while (left<right){
            int mid = (left + right)/2;
            if (tmp >= arr[mid])
                left = mid+1;
            else
                right=mid;
        }
        for (int j = i; j>left; --j){
            swap(&arr[j-1], &arr[j]);
        }
    }
}
```

Efficiency: Best case: `O(nlogn)`, `O(n^2)`, `O(n^2)`

$$\sum_{i=1}^{n-1}(O(\log i) + O(i)) = O(n^2) = \text{ number of comparisons } + \text{ number of swaps}$$

20 Exchange Sorting

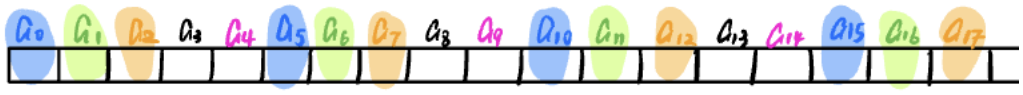
Algorithms that sort by exchanging pairs of items until the sequence is sorted. It may exchange adjacent elements as well as widely separated ones.



20.1 Shell Sort

The idea with shell sort is to make big strides between items, i.e., apply insertion sort to every Gth item

The idea is to start with subsets where the items are g apart. We then use insertion or bubble sort to sort just those items. then, we pick a smaller gap and repeat steps above. The last gap will be 1, and should have very few exchanges.



Start by defining gap $g = \frac{N}{3} + 1$, and we reduce by $g' = \frac{g}{3} + 1$. We sort each sublist $[k], [k+g], [k+2g]$, etc. using bubble or inserting sort

Gap = 5	a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11
Input	92	96	12	54	9	16	98	81	46	72	18	36
First	92					16					18	
Sorted	16					18					92	
Second		96					98					36
Sorted		36					96					98
Third			12					81				
Sorted			12					81				
Fourth				54					46			
Sorted				46					54			
Fifth					9					72		
Sorted					9					72		
Result	16	36	12	46	9	18	96	81	54	72	92	98

Gap = 3	16	36	12	46	9	18	96	81	54	72	92	98
First	16			46			96			72		
Sorted	16			46			72			96		
Second		36			9			81			92	
Sorted		9			36			81			92	
Third			12			18			54			98
Sorted			12			18			54			98
Result	16	9	12	46	36	18	72	81	54	96	92	98

Gap = 1												
	16	9	12	46	36	18	72	81	54	96	92	98
	16											
	9	16										
	9	12	16									
	9	12	16	46								
	9	12	16	36	46							
	9	12	16	18	36	46						
	9	12	16	18	36	46	72					
	9	12	16	18	36	46	72	81				
	9	12	16	18	36	46	54	72	81			
	9	12	16	18	36	46	54	72	81	96		
	9	12	16	18	36	46	54	72	81	92	96	
	9	12	16	18	36	46	54	72	81	92	96	98

```
void shellSort(int array[], int n){
    for (int interval = n/2; interval>0; interval/=2){
        int temp = array[i];
        int j;
        for (j = 1; j>= interval && array[j-interval]>temp;
        ↪  j-=interval) {
            array[j]=array[j-interval];
        }
        array[j]=temp;
    }
}
```

Efficiency: Best case: `O(Nlog N`, worse case: `O(n^2)`

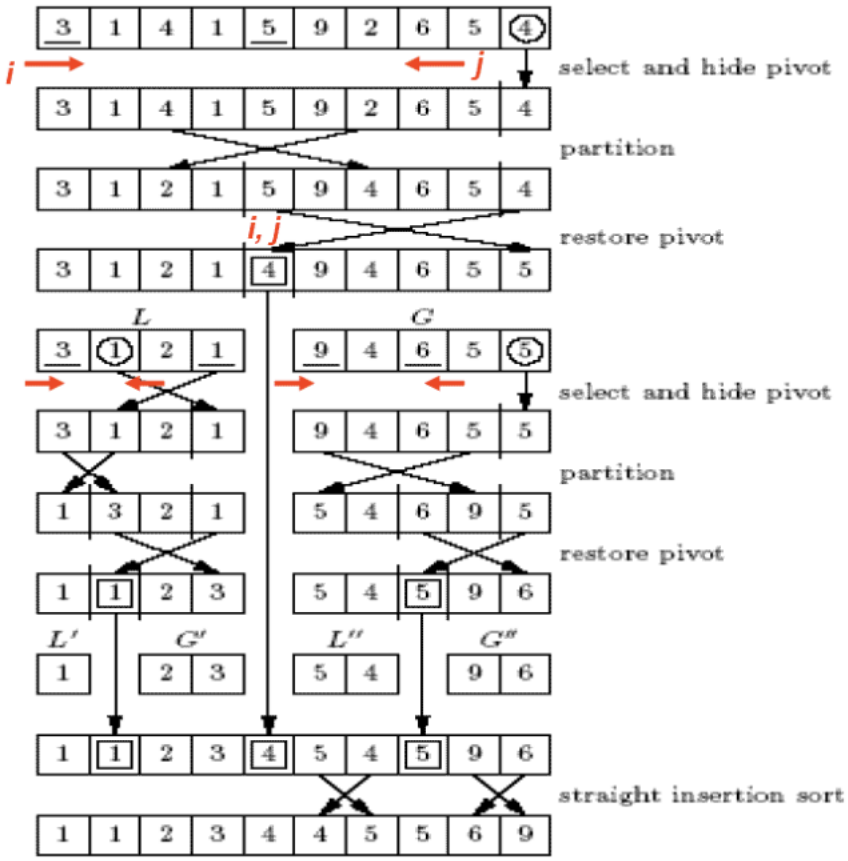
20.2 Quick Sort

Quick sort is a randomized sorting algorithm based on the *divide-and-conquer* paradigm:

Divide: pick a random element x (called pivot) and partition the sequence of data S into L elements less than x, E elements equal to x, G elements greater than x

Recur: sort L and G

Conquer: join L, E and G




```
void quicksort(int arr[]; int first; int last){
    int i,j,pivot,temp;
    if (first>last){
        pivot = first;
        i=first;
        j=last;
        while(i<j){
            while(arr[i]<=arr[pivot] && i<last)
                i++;
            while(arr[j]>arr[pivot])
                j--;
            if(i<j){
                temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
        temp = arr[pivot];
        arr[pivot]=arr[j];
        arr[j]=temp;
        quicksort(arr,first,j-1);
        quicksort(arr,j+1,last);
    }
}
```

Efficiency: Average case: $O(n \log n)$. Best case: $O(n \log n)$. Worst case: $O(n^2)$

21 Distribution Sorting

Distribution sorting *does not compare* the elements to be sorted. It requires that we know something about the basis set from which the elements to be sorted are drawn.

21.1 Bucket Sort

For example, an array with values between 0 and 25 can have its elements separated into buckets with ranges. For example, 9 and 8 would be in bucket 5-10.

```
int getBucketIndex(int value){
    int INTERVAL = 5; // determined based on bucket bounds
    return value/ INTERVAL;
}

void BucketSort(int arr[]){
    int i, j;
    struct Node **buckets;
    // memory allocate for NBUCKET
    for (i = 0; i < NBUCKET; ++i){
        buckets[i]=NULL;
    }

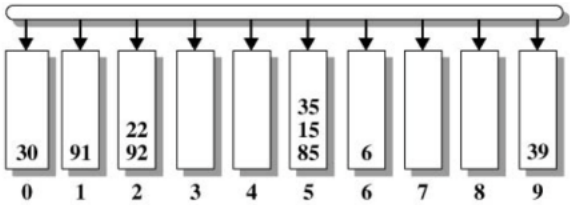
    for (i = 0; i<NARRAY; ++i){
        struct Node * current;
        int pos = getBucketIndex(arr[i]);
        current = (struct NOde *)malloc(sizeof(struct Node));
        current -> data = arr[i];
        current -> next = buckets[pos];
        buckets[pos]=current;
    }
    for (i = 0; i < NBUCKET; ++i) {
        buckets[i] = InsertionSort(buckets[i]);
    }
    // Put sorted elements on arr
    for (j = 0, i = 0; i < NBUCKET; ++i) {
        struct Node *node;
        node = buckets[i];
        while (node) {
            arr[j++] = node->data;
            node = node->next;
        }
    }
}
```

Efficiency: Worse case: $O(n^2)$. Best case: $O(n+k)$. Average case: $O(n+k)$

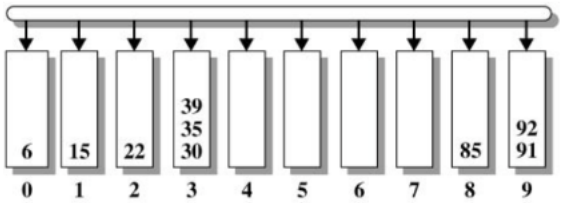
21.2 Radix Sort

This sorting algorithm sorts elements into buckets indexed from 0 to 9, using their digit place iteratively (ones, tens, hundreds)

List: [91, 6, 85, 15, 92, 35, 30, 22, 39]



After Pass 0: [30, 91, 92, 22, 85, 15, 35, 6, 39]



After Pass 1: [6, 15, 22, 30, 35, 39, 85, 91, 92]

```
void radixsort(int array[], int size){
    int max = getMax(array,size);
    for (int place = 1; max / place > 0; place*= 10){
        countingSort(array, size, place);
    }
}
```

```
void countingSort(int array[], int size, int place) {
    int output[size + 1];
    int max = (array[0] / place) % 10;

    for (int i = 1; i < size; i++) {
        if (((array[i] / place) % 10) > max)
            max = array[i]; }

    int count[max + 1];
    for (int i = 0; i < max; ++i)
        count[i] = 0;

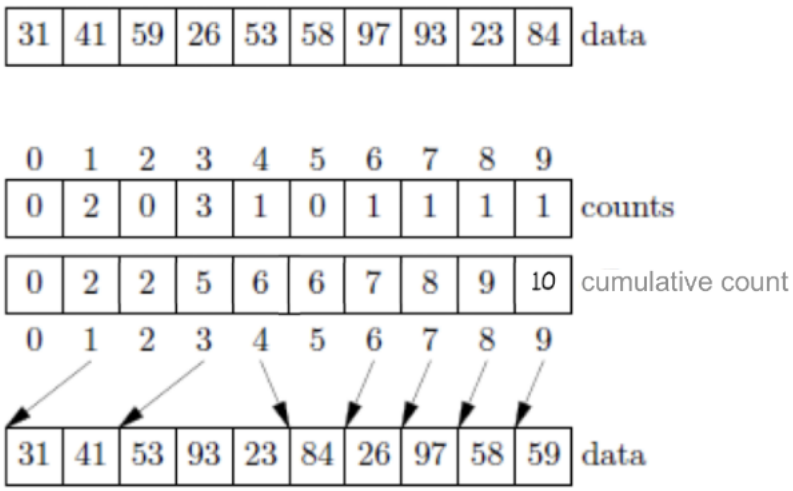
    // Calculate count of elements
    for (int i = 0; i < size; i++)
        count[(array[i] / place) % 10]++;

    // Calculate cumulative count
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Place the elements in sorted order
    for (int i = size - 1; i >= 0; i--) {
        output[count[(array[i] / place) % 10] - 1] = array[i];
        count[(array[i] / place) % 10]--; }

    for (int i = 0; i < size; i++)
        array[i] = output[i];
}
```

Efficiency: $O(n)$

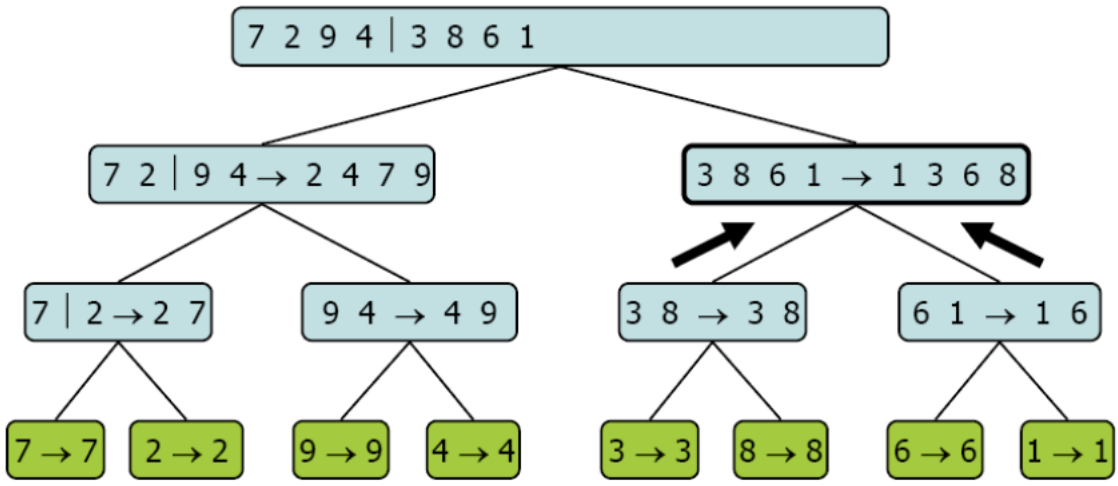


22 Merge Sorting

Divide and conquer is a problem-solving technique that makes use of recursion

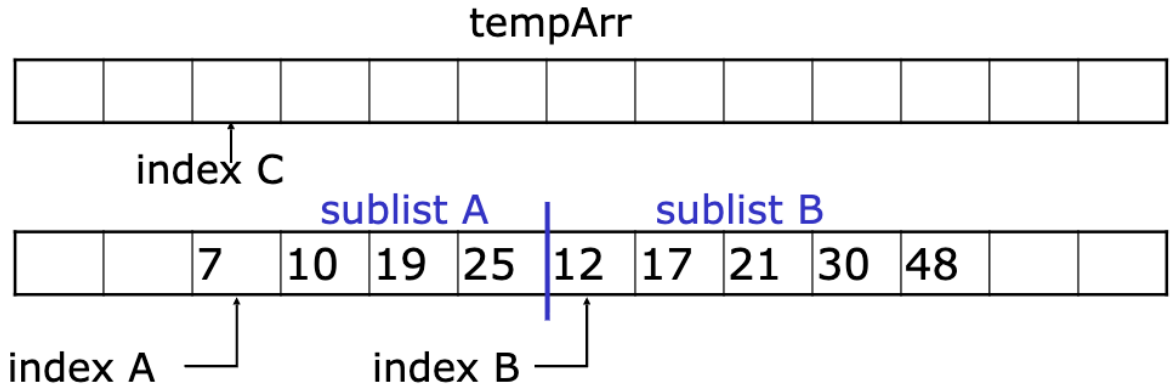
22.1 Divide and Conquer

We can **divide** the elements into two groups of roughly equal size, recursively, until a group is so small it can be sorted. Then, **combine** each pair of sorted groups into one large sorted list



22.1.1 Merging Process

Consider two sorted sublists:



set indices for the first element in sublist A, sublist B, and the tempArr respectively. Then, for example, consider 7 and 12, where 7 will go to `indexC` and then next comparison will go to `indexC+1`

```
void sort(int arr[]){
    // create tempArr
    // get size and allocate memory
    for (int i = 0; i<size; i++){
        tempArr[i]=arr[i];
    }
    msort(arr, tempArr, 0, size);
}
```

22.1.2 Partition Process

- Define midpoint

```
int midpt = (last + first)/2
```

- Each iteration involves two calls to msort, where `first+1<last`
- The partition stops when the size of sublist is 1, where `first + 1 = last`

```
void msort(int arr[], int tempArr[], int first, int last){
    if (first + 1 < last){
        int midpt = (last + first)/2;
        msort(arr, tempArr, first, midpt);
        msort(arr, tempArr, midpt, last);
        if (arr[midpt-1] <= arr[midpt]){
            return;
        }
        merge(arr, tempArr, first, midpt, last);
    }
}
```

```
void merge(int arr[], int tempArr[], int first, int midpt, int last){
    int indexA, indexB, indexC;
    indexA=first;
    indexB=midpt;
    indexC=last;
    while (indexA < midpt && indexB < last){
        if (arr[indexA]< arr[indexB]) {
            tempArr[indexC] = arr[indexA];
            indexA++;
        }
        else {
            tempArr[indexC] = arr[indexB];
            indexB++;
            }
        indexC++;
    }
    while(indexA < midpt)  {
        tempArr[indexC] = arr[indexA];
        indexA++;
        indexC++;
        }
    while(indexB < last)  {
        tempArr[indexC] = arr[indexB];
        indexB++;
        indexC++;
        }
    for(int i=first; i< last; i++)  {
        arr[i] = tempArr[i];
    }
}
```

Efficiency: $O(n \log n)$

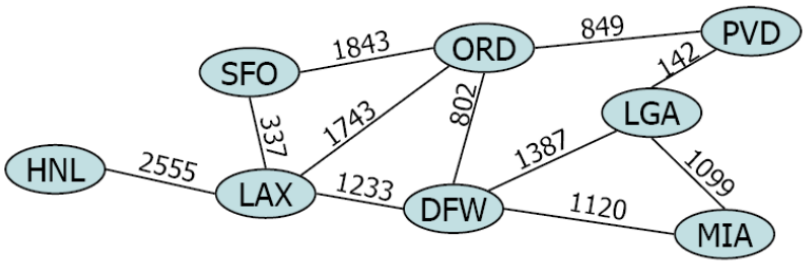
23 Graphs

A graph is a data structure that can represent a finite number of elements and relationships of some kind between those elements

A graph is a pair (V, E) where V is a set of nodes, called **vertices**, and E is a collection of pairs of vertices, called **edges**

A vertex represents an airport and stores the three-letter airport code

An edge represents a flight route between two airports and stores the mileage of the route

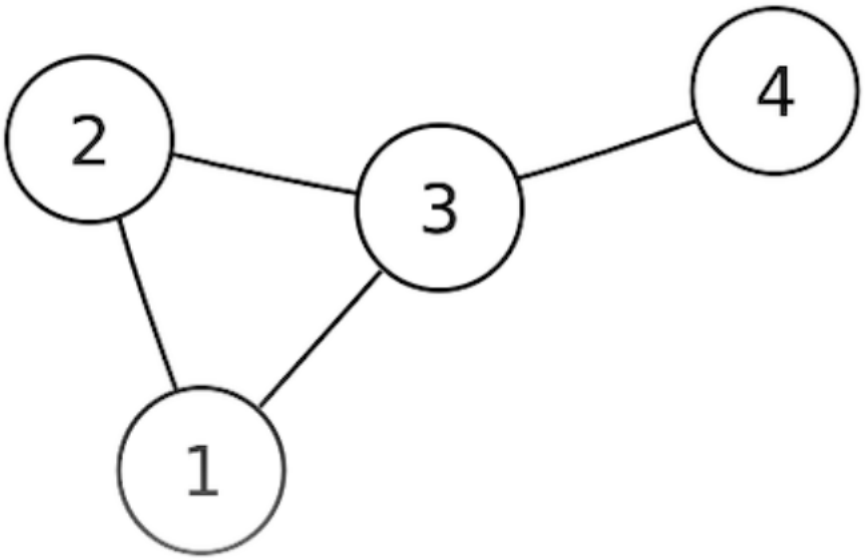


23.1 Types of Graphs

Undirected Graphs follow these properties:

- The first component, V, is a finite, non-empty set. The elements of V are called the vertices of G
- The second component, E, is a finite set of sets. Each element of E is a set that is comprised of exactly two (distinct) vertices. The elements of E are called the edges of G

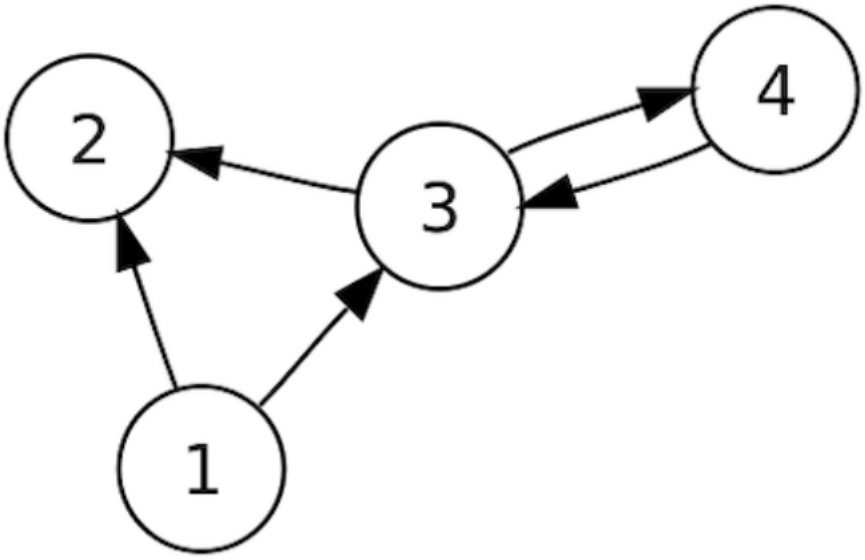
Example: If $E = \{(1, 2)\}$ is given, it can go either way $1 \leftrightarrow 2$



Directed Graphs follow these properties:

- The first component V is a finite, non-empty set. The elements of V are called the vertices of G
- The second component E is a finite set of ordered pairs of vertices. The elements of e are called the edges of G

Example: If $E = \{(1, 2)\}$ is given, you can only go $1 \to 2$. However, if $E = \{(3, 4), (4, 3)\}$ is given, then we can go $3 \leftrightarrow 4$

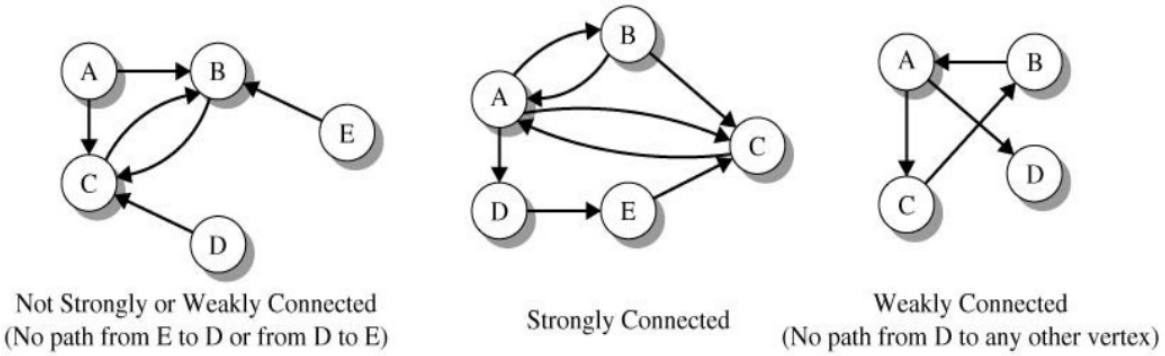


23.2 Connectivity

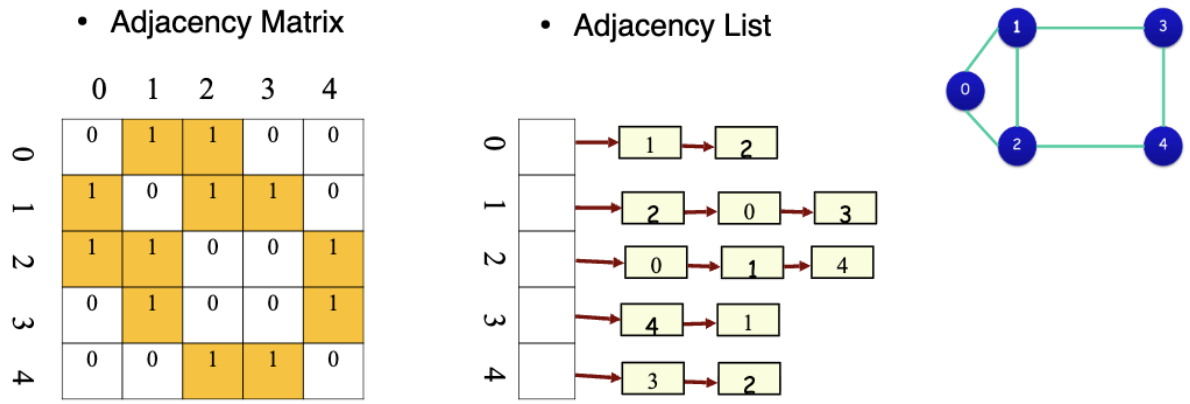
An *undirected graph* is **connected** if there is a path between **every pair** of vertices

A *directed graph* is **strongly connected** if there is a path from any vertex to any other vertex

A *directed graph* is **weakly connected** if, for each pair of vertices (a, b), there is a path $a \to b$ or $b \to a$



23.3 Representation



For the adjacency matrix, consider:

$$A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency matrices are best used if you have lots of edges between vertices but few vertices.

For the adjacency list, consider:

$$\forall v_i \in V, \{w : (v_i, w) \in A(v_i)\}$$

Adjacency lists are best used if you have few edges between vertices and lots of vertices.

Adjacency matrices are different whether the graph is directed vs undirected. An undirected graph is symmetric on its diagonal.

```c
void create_graph()
{
    int n, adj[100][100];
    int count, max_edge, origin, destin;
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edge = n*(n-1);
    for(count=1; count<=max_edge; count++)
    {
        printf("Enter edge %d( -1 -1 to quit ) : ", count);
        scanf("%d %d",&origin,&destin);
        if((origin == -1) && (destin == -1))
            break;
        if(origin>=n || destin>=n || origin<0 || destin<0)
        { printf("Invalid edge!\n");
            count--; }
        else
        { adj[origin][destin] = 1; }
    }
}
```

24 Graph Traversals

A graph has two different ways to be traversed.

Breadth-First Search: explores all vertices at the current distance (or depth) from the starting node before moving on to vertices at the next distance level

Depth-first Search: explores a graph by visiting a vertex and then recursively exploring as far along each branch as possible before backtracking

24.1 Breadth-First Search

This is similar to a level-by-level manner as in level order traversal of trees. BFS uses a queue to keep track of which vertex to visit next.

1. Start from a selected source vertex
2. Visit all its direct neighbours first (one edge away from vertex)
3. Then, move on to the next level, visit all neighbours of those nodes that haven't been visited yet
4. Continue this process until all reachable vertices have been visited

Implementation:

```
#define MAX 100

#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);

int queue[MAX], front = -1, rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();
```

```
void create_graph()
{
  int count, max_edge, origin, destin;
  printf("Enter number of vertices : ");
  scanf("%d",&n);
  max_edge = n*(n-1);
  for(count=1; count<=max_edge; count++)
  {
    printf("Enter edge %d( -1 -1 to quit ) : ",count);
    scanf("%d %d",&origin,&destin);
    if((origin == -1) && (destin == -1))
      break;
    if(origin>=n || destin>=n || origin<0 || destin<0)
    {    printf("Invalid edge!\n");
         count--;  }
    else
    {   adj[origin][destin] = 1; }
  }}
```

```
void BF_Traversal()
{
  int v;

  for(v=0; v<n; v++)
    state[v] = initial;

  printf("Enter Start Vertex for BFS: \n");
  scanf("%d", v);
  BFS(v);
}
```

```
void BFS(int v)
{
  int i;
  enqueue(v);
  state[v] = waiting;
   while(!isEmpty_queue())
  {
    v = dequeue( );
    printf("%d ",v);
    state[v] = visited;
     for(i=0; i<n; i++)
    {
      if(adj[v][i] == 1 && state[i] == initial)
      {
        enqueue(i);
        state[i] = waiting;
      } }}
  printf("\n");  }
```

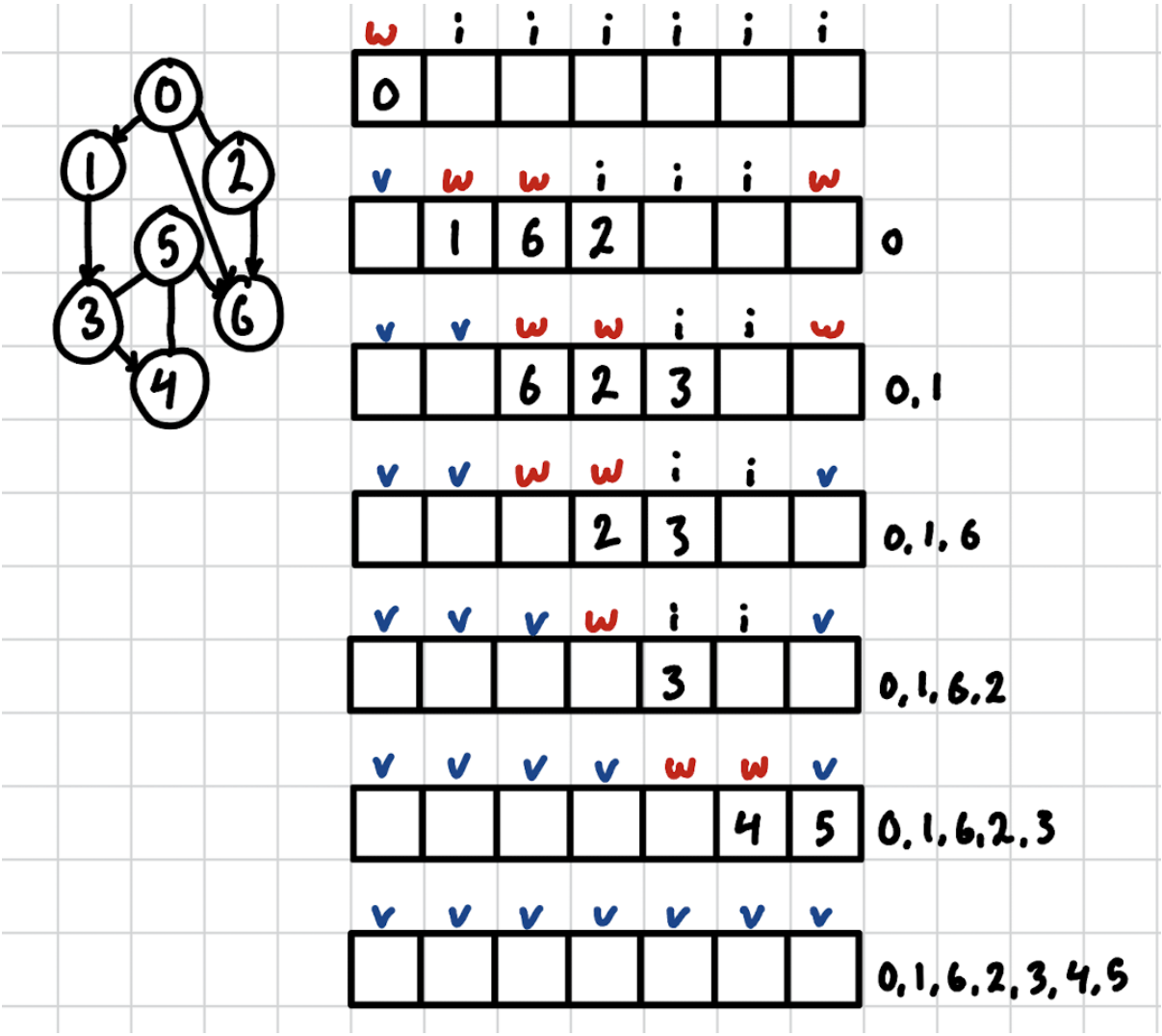
24.1.1 BFS Running Time

For an adjacency matrix representation, the number of comparisons are: `V iterations * V neighbor checks = V^2 checks`

Therefore, the worst-case running time with an adjacency matrix is $O(|V|^2)$

For an adjacency list representation, there are $|V|$ number of vertices and $|E|$ number of edges.

Therefore, the total cost is $O(|V| + |E|)$

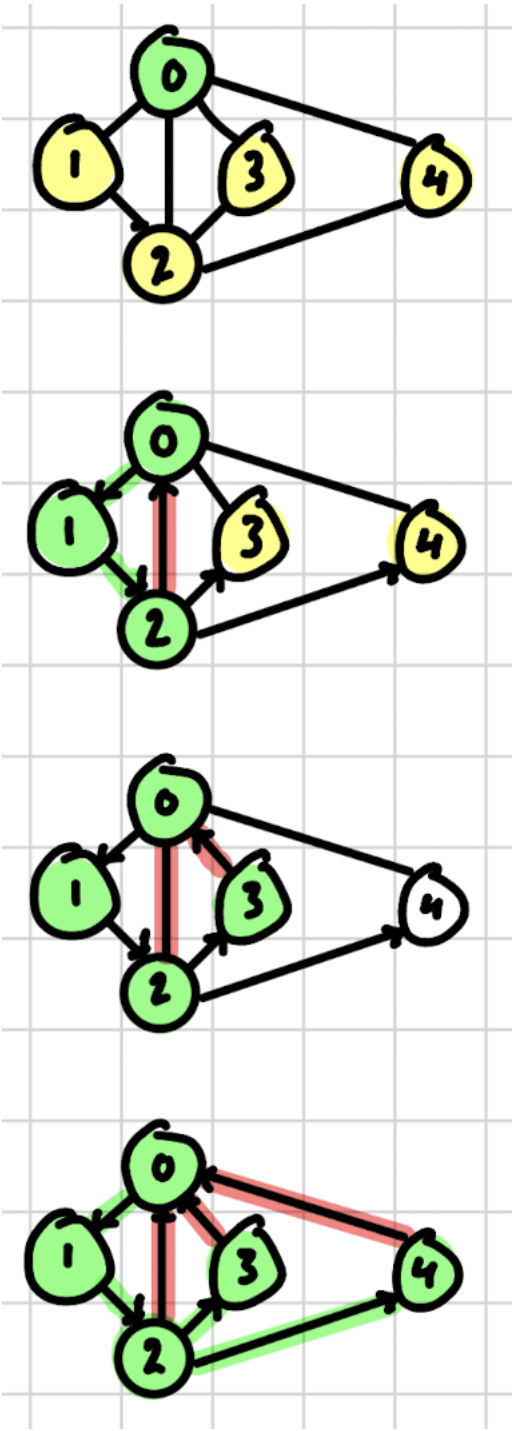


24.2 Depth-First Search

DFS keeps moving forward until it hits a dead end or a previously-visited vertex, then it backtracks and tries another path

- the traversal must visit every vertex at most once
- we keep track of the visited nodes
- a DFS traversal of a graph visits a vertex and
  - recursively visit all of the vertices adjacent to that node
  - only follows edge that lead to unvisited vertices

DFS is more memory efficient than breadth first search as you can backtrack sooner. However, DFS cannot guarantee that it will find either shortest or longest paths



Implementation:

```
void DFS(int);
int G[10][10], visited[10], n;
//n is # of vertices and graph is
sorted in array G[10][10]

void DFS(int i)
{
   int j;
   printf("\n%d",i);
   visited[i]=1;

   for(j=0;j<n;j++)
     if(!visited[j]&&G[i][j]==1)
        DFS(j);
}
```

IDEA:

```
n ← number of nodes

Initialize visited[ ] to false (0)
for(i=0;i<n;i++)
   visited[i] = 0;

void DFS(vertex i)   %DFS starting from i
{
   visited[i]=1;
   for each w adjacent to i
      if(!visited[w])
         DFS(w);
}
```

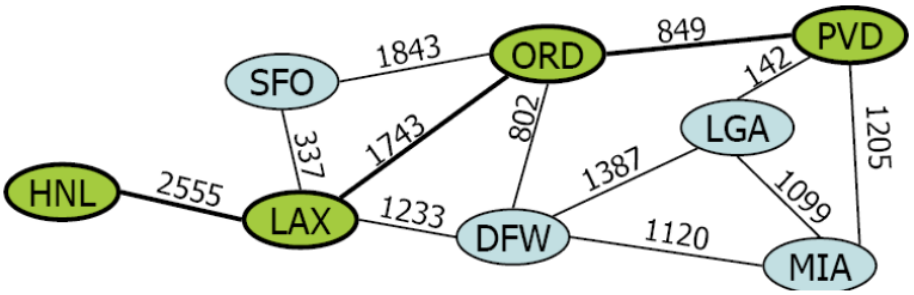
24.2.1 DFS Running Time

Same as BFS, DFS with adjacency matrix is $O|V|^2$, and with adjacency list is $O(|V|+|E|)$

25 Shortest Path Problems

Definition: A weighted graph assigns a weight to each edge of the graph.

Example: Find the shortest path between Providence and Honolulu.



In the example above, we must let $C(v_i, v_j)$ be the weight of the edge connecting both vertices, and a *path* is a non-empty sequence of vertices where each consecutive pair of vertices is connected by an edge.

The weighted path length of path P is given by:

$$\sum_{i=1}^{k-1} C(v_i, v_{i+1})$$

25.1 Dijkstra's Algorithm

Definition: The **distance** of a vertex v from a vertex s is the length of a shortest path between s and v. The vertex s is the Single Source

Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s with the following assumptions:

- the graph is connected;
- the edges are directed or undirected;
- the edge weights are nonnegative

Idea: we grow a "cloud" of vertices, beginning with the given start vertex s and eventually covering all the vertices.

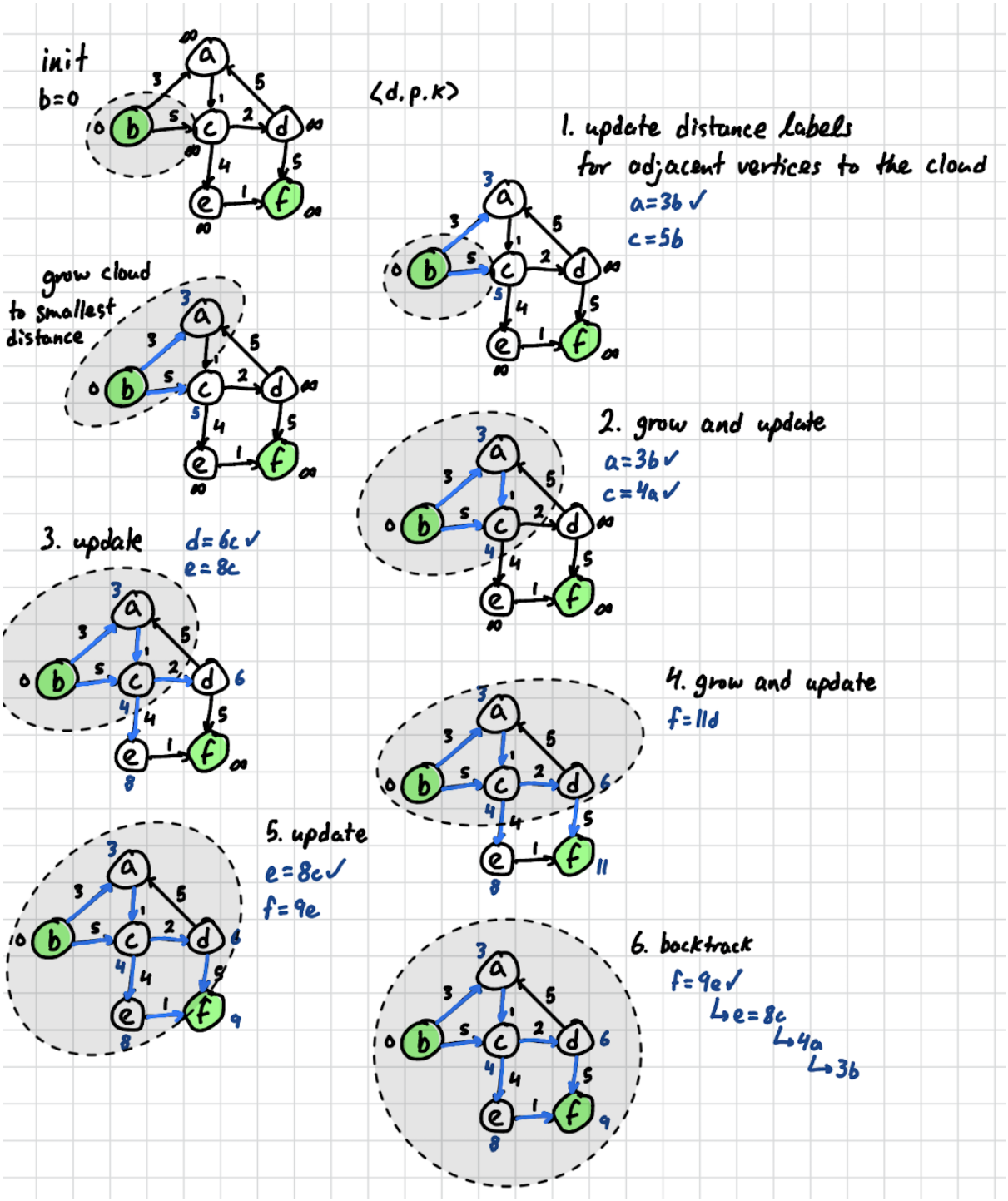
1. We store with each vertex u a label $d(u)$, representing the distance of u from s in the subgraph consisting of the cloud and its adjacent vertices
2. We add to the cloud, i.e., the vertex z outside the cloud with the *smallest* distance label, $d(z)$. We update the labels of the vertices adjacent to z

Edge Relaxation: Consider an edge $e = (u, z)$, where z is not in the cloud. The relaxation of edge e updates the distance $d(z)$ as follows:

$$d(z) \leftarrow min\{d(z), d(u) + weight(e)\}$$

Information: Each vertex u takes three pieces of information $<$ distance, predecessor, flag $>$

- distance is the length of the shortest known path from s to u
- predecessor of vertex v on the shortest path from s to u
- flag is a boolean-valued k indicates that the shortest path to vertex u is known



Implementation:

```
void Dijkstra(int Graph[MAX][MAX], int n, int start) {
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;

    // Creating cost matrix (weights)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (Graph[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = Graph[i][j];

    for (i = 0; i < n; i++) {
        distance[i] = cost[start][i];
        pred[i] = start;
        visited[i] = 0;
    }

    distance[start] = 0;
    visited[start] = 1;
    count = 1;

    while (count < n - 1) {
        mindistance = INFINITY;
        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }

        visited[nextnode] = 1;

        for (i = 0; i < n; i++)
            if (!visited[i])
                if (mindistance + cost[nextnode][i] < distance[i]) {
                    distance[i] = mindistance + cost[nextnode][i];
                    pred[i] = nextnode;
                }

        count++;
    }
}
```

25.2 Subgraph and Spanning Tree

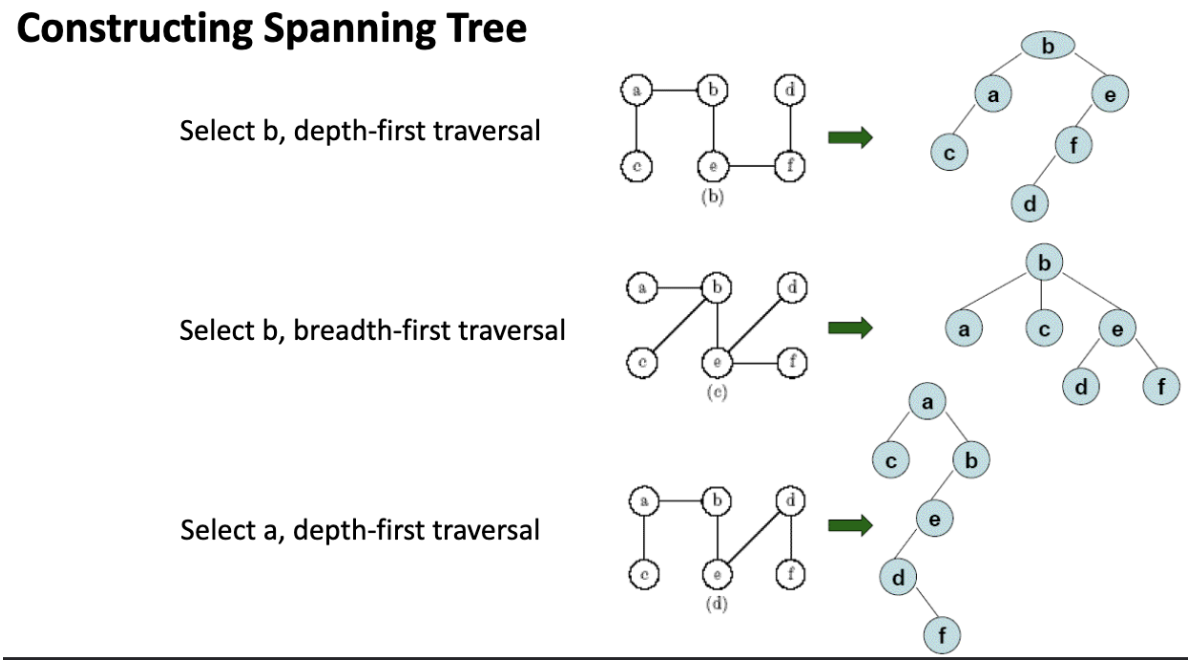
Definition: A *subgraph* of a graph $G = (V, E)$ is any graph $G' = (V', E')$ such that $V' \subseteq V$, and $E' \subseteq E$

Definition: Consider a connected, undirected graph $G = (V, E)$. A *spanning tree* of G is

a subgraph of G, say $T = (V', E')$, with: $V' = V$; T is connected; and T is acyclic

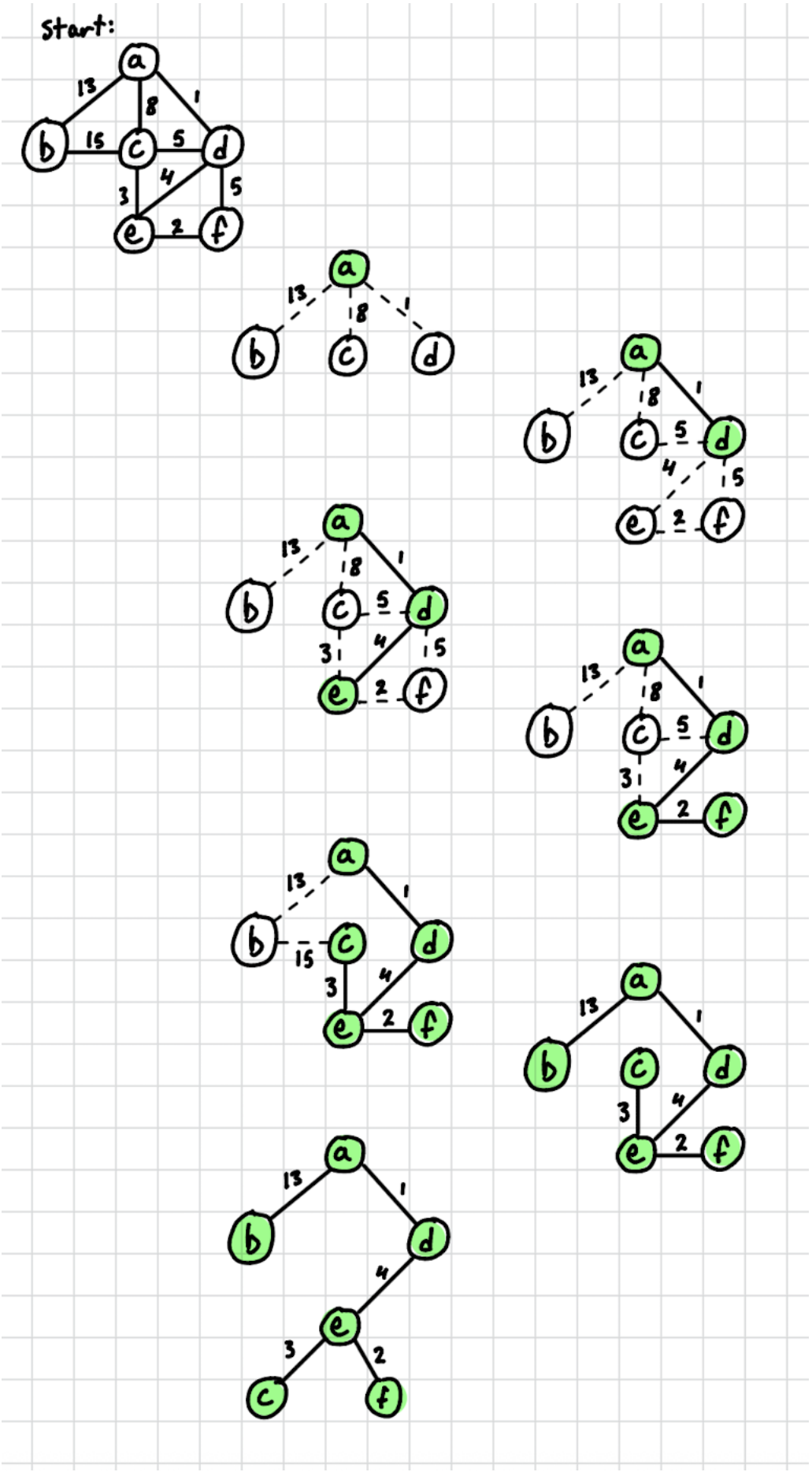
A Spanning Tree is minimum cost if it has the smallest total cost:

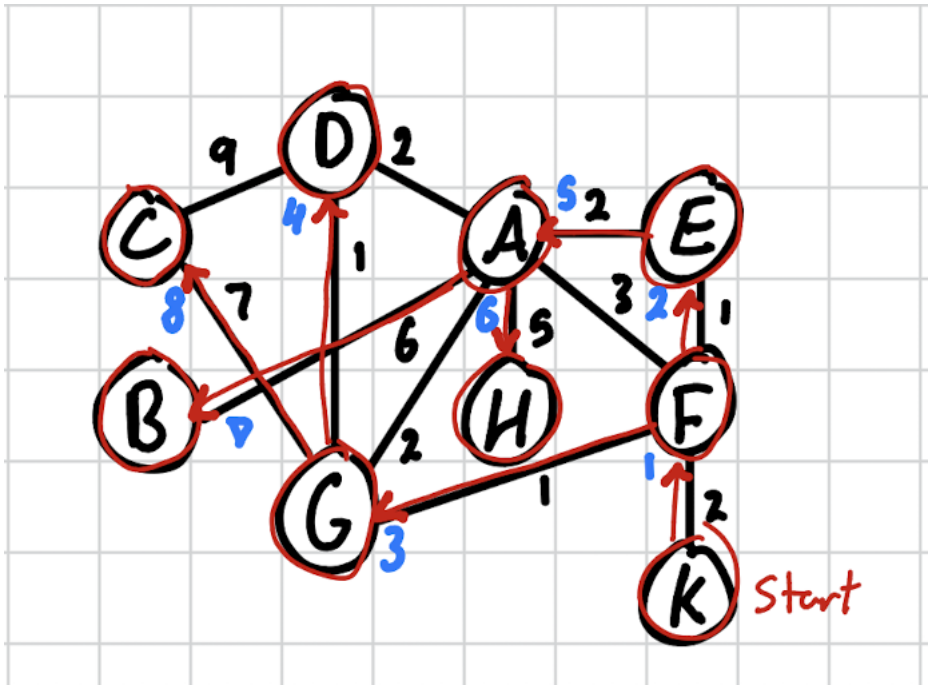
$$\sum_{(v,w)\in E} C(v, w)$$



25.3 Prim's Algorithm

- A minor variation of Dijkstra's Algorithm
- Construct the minimum-cost spanning tree of a graph by selecting edges from the graph one-by-one and adding those edges to the spanning tree
- At each step, select an edge with the smallest edge weight that connects the tree to a vertex not yet in the tree


Start:
a
b
c
d
e
f
13
8
1
15
5
4
3
5
2


D
C
A
E
B
H
F
G
K
Start

26 ELEC 278 Final Study

Time Complexity

AVL Trees

- Q: In an AVL tree, which condition must always hold after every insertion or deletion at every node v? A: $|height(v_{left}) - height(v_{right})| \leq 1$
- Q: In an AVL tree insertion that creates a LR imbalance at node z, which rotation sequence restores balance while preserving the BST order? A: Left rotation at z's left child, then right rotation at z
- Q: What is the worst-case height $h(n)$ of an AVL tree with n nodes? A: $h(n) = \Theta(\log n)$
- Q: During deletion in an AVL tree, why might rebalancing continue upward beyond the first rotation point? A: Deletion can reduce subtree height, potentially causing a new imbalance at ancestors even after a local fix
- Q: Suppose a node z has $BF(z) = -2$ and its right child y has $BF(y) = +1$. Which rotation sequence correctly restores AVL balance? A: Right rotation at y, then left rotation at z
- Q: Every AVL tree is also a binary search tree. A: True

Max-heaps

- Q: In a max heap implemented as a binary tree, which property must always hold for every node? A: Every node's key is greater than or equal to the keys of its children
- Q: You have a max heap stored in an array using 0-based indexing. For a node at index i, what is the index of its left child (assuming it exists)? A: $2i + 1$
- Q: What is the worst-case time complexity of inserting a new element into a max heap containing n elements? A: $O(\log n)$
- Q: A level-order traversal of a max-heap always visits the node sin non-increasing order. A: False

Hash Tables

- Q: The worst case running time for insertion in a closed hash table with size m is $O(m)$. A: True

Trees

- Heaps are complete binary trees
- Every full binary tree is also a complete tree

Graphs

- Trees are special types of graphs (connected, acyclic)
- Adjacency matrix worst-case is $O(|V|^2)$
- Adjacency list worst-case is $O(|V| + |E|)$
- BFS is faster than Dijkstra's for finding the shortest path from one vertex to another in an unweighted graph
- We use a Queue for a BFS
- We use Prim's to construct the minimum-cost spanning tree of a graph
- We use Dijkstra's to find a single-source shortest path

Sorting

- Insertion is the most efficient to sort a nearly sorted array

Running-Time

- Q: If algorithm A has running time $7n^2$ and B has $2n^2 + n$ then they have the same asymptotic time complexity (Big Oh). A: True

Real-World Questions

- Q: Give examples where a double linked list would be better than a single linked list? A: Browser navigation (undo and redo) and Media players (next song vs. playback)
- Q: Give examples where a circular linked list would be better than a regular linked list? A: Media player repeat mode and Online card games
- Q: Why do real-world software applications treat the average case time complexity of hash tables as their actual behaviour, but not for BSTs? A: The worst-case of trees is common, and relying on the average-case is often inaccurate. For a hash table, the worst-case only occurs if all keys collide. A good hash function prevents this and makes the worst case very uncommon.