CSCI E-50 WEEK 5

TERESA LEE
[teresa.lee@alumni.utoronto.ca]
October 2, 2017

Today

- Structs
- Linked lists
- Hash tables
- Tries
- Stacks & Queues

Structures

- Structures provide a way to unify several variables of different data types into a single new variable type which can be assigned its own type name.
- We use structs to group together elements of a variety of data types that have a logical connection.
- A structure is like a “super variable.”
- The variables contained within a field are sometimes called "attributes," or "fields," or "members."

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    bool automatic;
};
```

Where have you seen structures?

Examples of Structures

```
typedef struct
{
    BYTE rgbtBlue;
    BYTE rgbtGreen;
    BYTE rgbtRed;
} __attribute__((__packed__))
RGBTRIPLE;
```

```
typedef struct
{
    WORD bfType;
    DWORD bfSize;
    WORD bfReserved1;
    WORD bfReserved2;
    DWORD bfOffBits;
}
__attribute__((__packed__))
BITMAPFILEHEADER;
```

```
typedef struct
{
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} __attribute__((__packed__))
BITMAPINFOHEADER;
```

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

Structures

- We typically define structures inside of a .h file to abstract them away and make them able to be used by other programs as well.
- We typically define our structures by *defining* a new custom type with the typedef keyword.
- Individual fields of the structure are accessed using the dot operator (.)

Structures

- Structures can be dynamically allocated as well, using malloc().

```
// declaration
struct car *mycar = malloc(sizeof(struct car));
Car_info *mycar = malloc(sizeof(car_info));
```

- “.”
  -
- In order to access the fields of a dynamically allocated structure, we must first dereference the pointer and then access the field.
  - (*struct).field
  - OR! struct->field

Structures

```
// declaration
struct car *mycar = malloc(sizeof(struct car));

// field access
mycar->year = 2011;
strcpy(mycar->plate, “CS50”);
mycar->automatic = false;
```

- The type name “struct car” is a bit inconvenient. That line dynamically allocating a single structure got quite long because I had to type “struct car” twice.
- C provides a way to alias data types to give them a different name. This power comes in handy most commonly in the context of structures, but it has general applicability.
- The keyword that handles this is typedef.

Structures

Instead of...

```
// definition
struct car
{
     string name;
     int year;
}
car_info;

// declaration
struct car *mycar = malloc(sizeof(struct
car));
```

You can…

```
// definition
typedef struct car
{
     string name;
     int year;
}
car_info;

// declaration
car_info *mycar = malloc(sizeof(car));
```

Let's Look at an Example

Example 1

Example 2

Example 3

Linked List

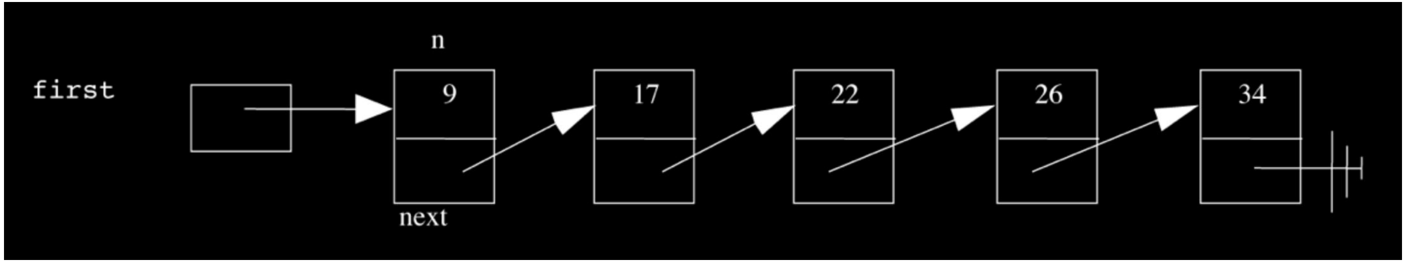
- A “chain” of nodes - a combination of structs and pointers.
- A linked list **node** is a special type of struct with two fields:
  - Data of some type
  - A pointer to another linked list node.
- How is it different from array?

Linked list

Tasks

- Create a linked list
- Search an element
- Insert a node
- Delete the entire list
- Delete a single element

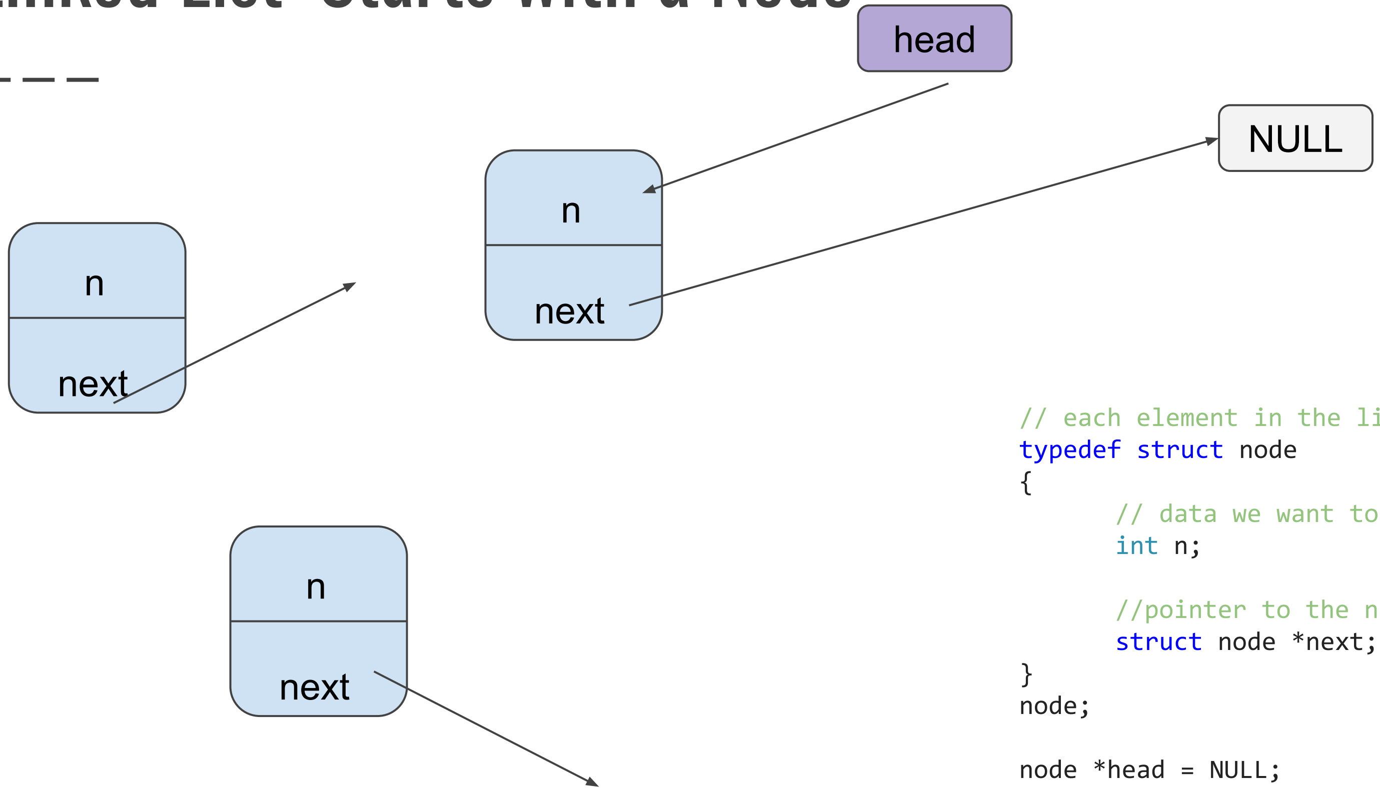
```c
// each element in the linked list
typedef struct node
{
        // data we want to store
        int n;

        //pointer to the next element in the list
        struct node *next;
}
node;
```



Linked List: Starts with a Node

1. Dynamically allocate space for a new node.
2. Check to make sure you didn’t run out of memory.
3. Initialize the value field.
4. Initialize the next field (specifically, to NULL).
5. Return a pointer to your newly created node.

Linked List: Starts with a Node




```c
// each element in the linked list
typedef struct node
{
        // data we want to store
        int n;

        //pointer to the next element in the list
        struct node *next;
}
node;

node *head = NULL;
```

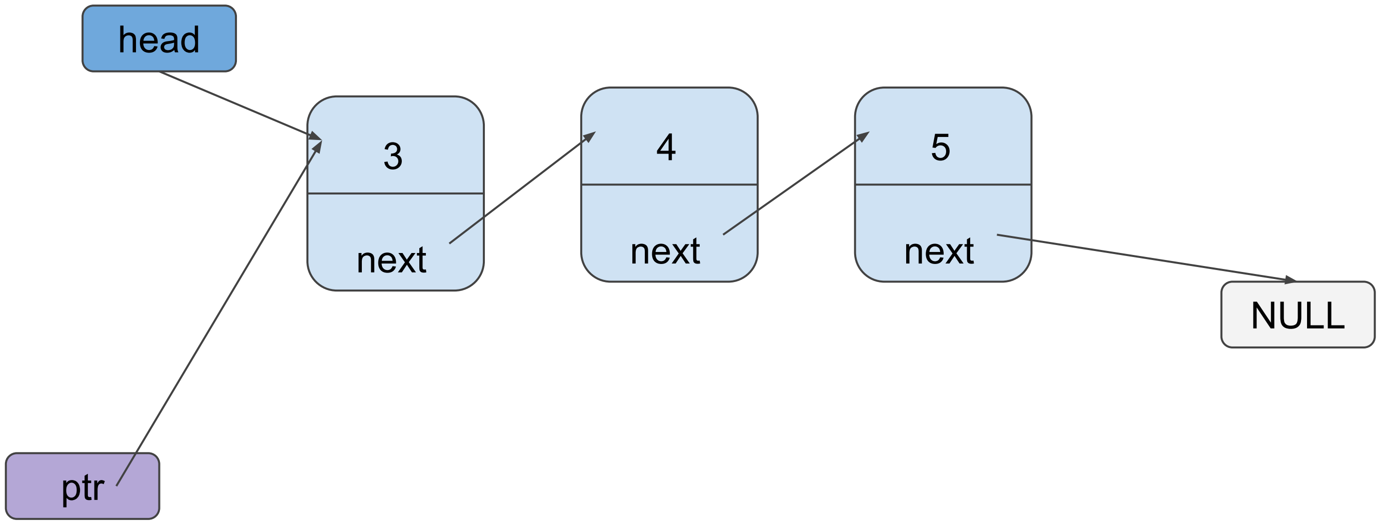
Let’s Look at an Example

Example 5

Linked List: Search an Element

1. Create a traversal pointer pointing to the list’s head (first element).
2. If the current node’s value field is what we’re looking for, return true.
3. If not, set the traversal pointer to the next pointer in the list and go back to the previous step.
4. If you’ve reached the last element of the list, return false.

Linked List: Search an Element



Let's Look at an Example

```
Example 6
```

Linked List: Search an Element

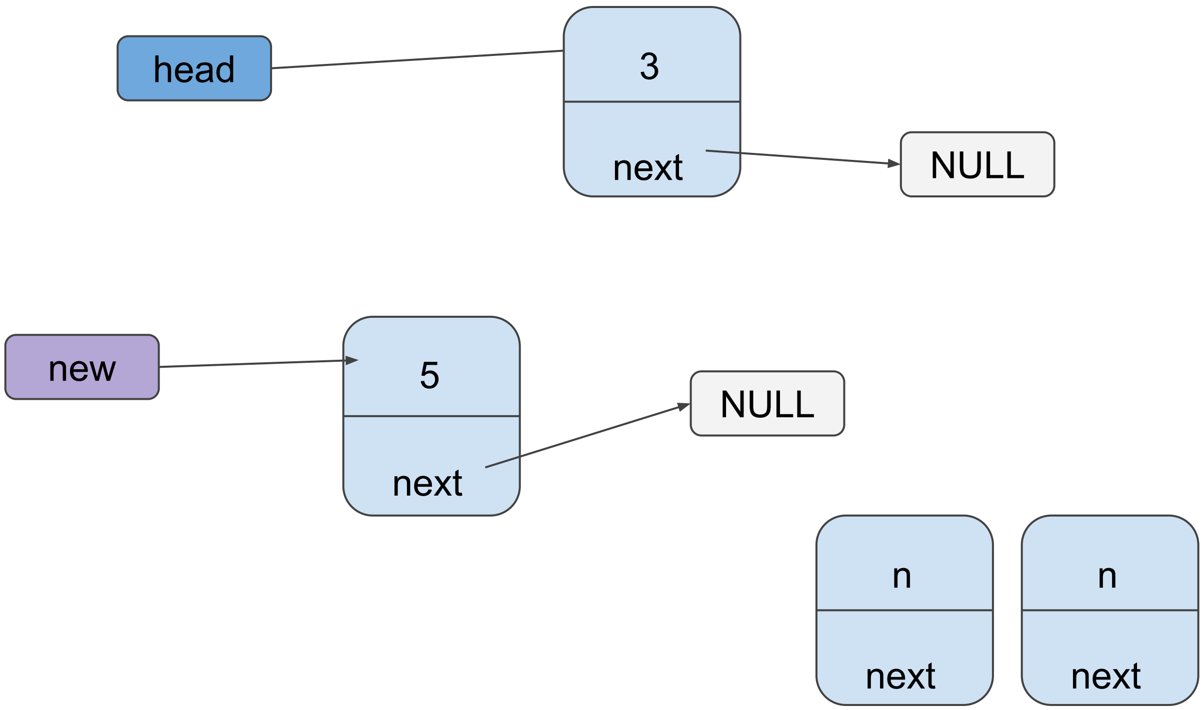
```
bool search(int n, node *list)
{
    // create a traversal pointer
    node *ptr = list;

    // traverse until the end of the list
    while (ptr != NULL)
    {
        // if current n field is what we are looking for, return true
        if (ptr->n == n)
        {
            return true;
        }

        // if not, set ptr to the next pointer in the list
        ptr = ptr->next;
    }
    return false;
}
```

Linked List: Insert a Node

---

1. Dynamically allocate space for a new linked list node.
2. Check to make sure we didn't run out of memory.
3. Populate and insert the node at the beginning of the linked list.
   a. So which pointer do we move first? The pointer in the newly created node, or the pointer pointing to the original head of the linked list?
   b. This choice matters!
4. Return a pointer to the new head of the linked list.

Linked List: Insert a Node



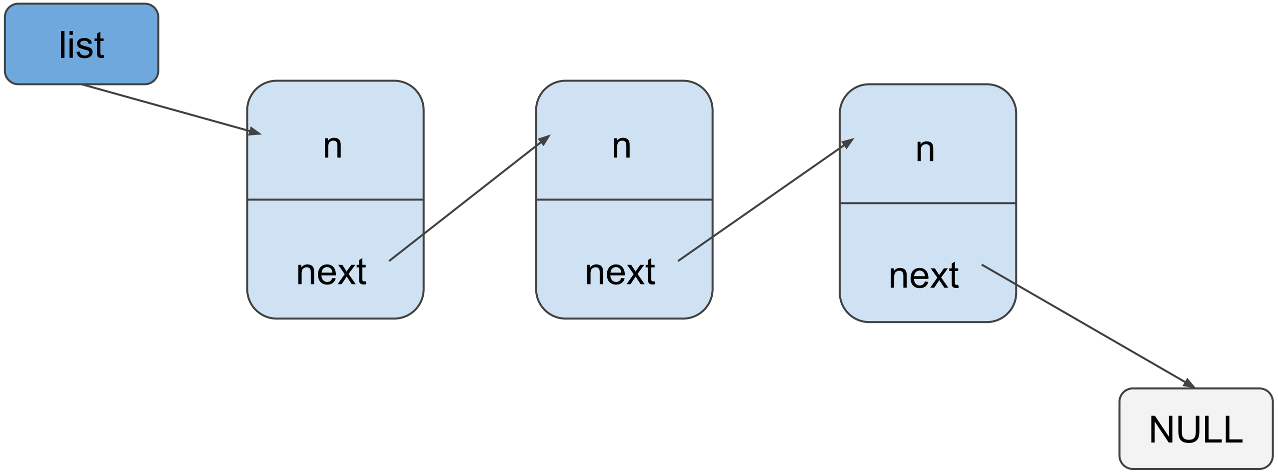
Let’s Look at an Example

Example 7

Linked List: Insert a Node

```
void insert(int n)
{
    //dynamically allocate space for a new node
    node *new = malloc(sizeof(node));

    //check to make sure we didn’t run out of memory
    if (new == NULL)
    {
         exit(1);
    }

    // initialize the n field
    new -> n = n;

    // initialize the next field
    new -> next = NULL;

    //check if the list exists
    if (head == NULL)
    {
        //create a pointer to the new list
        head = new;
    }
    else
    {
        // insert new node at head
        new->next = head;
        head = new;
    }
}
```

Linked List: Empty List



Linked List: Empty List

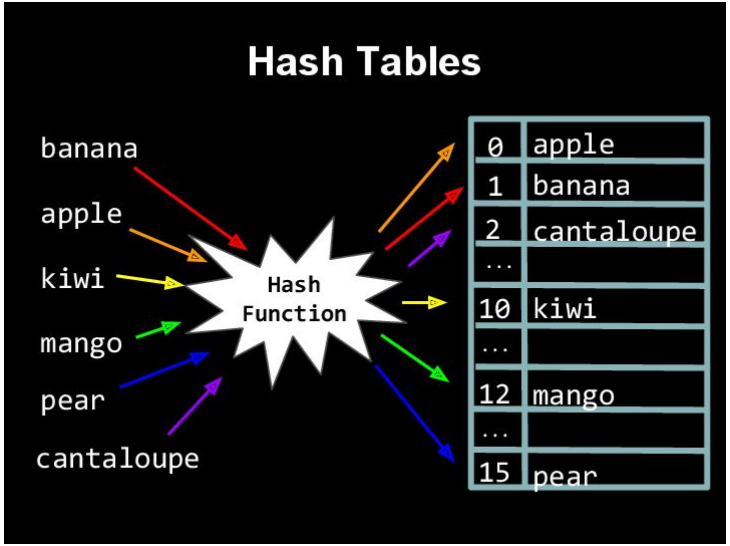
```
void empty_list(node *list)
{
    // if the list is empty, nothing else to do
    if (list == NULL)
    {
        return;
    }

    // empty the rest of the list
    empty_list(list->next);

    //free the current node
    free(list);
}
```

Linked List: Pros & Cons

- Pro: ability to grow and shrink
- Con: slower searches, insertions, deletions

Hash Table

- An "associative array" where the position of each element is decided by (associated with) the result of passing data into a *hash function*.
- Combines the random access of an array with the dynamism of a linked list.
- This means insertion deletion, and lookup can all tend toward Θ(1)! We're gaining the advantages of both, and mitigating the disadvantages.
- Cons? Not good at ordering or sorting data



Hash Table: Hash Function

A hash function describes where to insert a word and, when necessary, where to look up a word.

A good hash function should:

- Use only the data being hashed
- Use all of the data being hashed
- Be deterministic (return same value for same data)
- Uniformly distribute data
- Generate very different hash codes for very similar data

```
string hashtable [10];

unsigned int hash(char *str)
{
      Int sum = 0;
      for(int j = 0; str [j] != '\0'; j++)
      {
            sum += str [j];
      }

      return sum % HASH_MAX;
}
```

Problem?! What happens if there are more than one word resulting in the same hash code? COLLISION

Hash Table

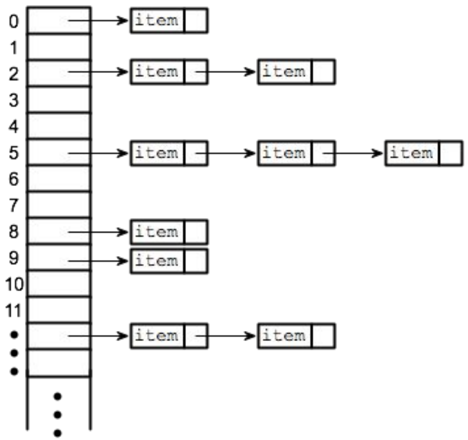
A solution to a collision?

- Linear probing: if a key hashes to the same index as a previously stored key, it is assigned the next available slot in the table
- Upperbound insertion, deletion, and lookup times have devolved to O(n), where n is the size of the table
- Linear probing may lead to clustering

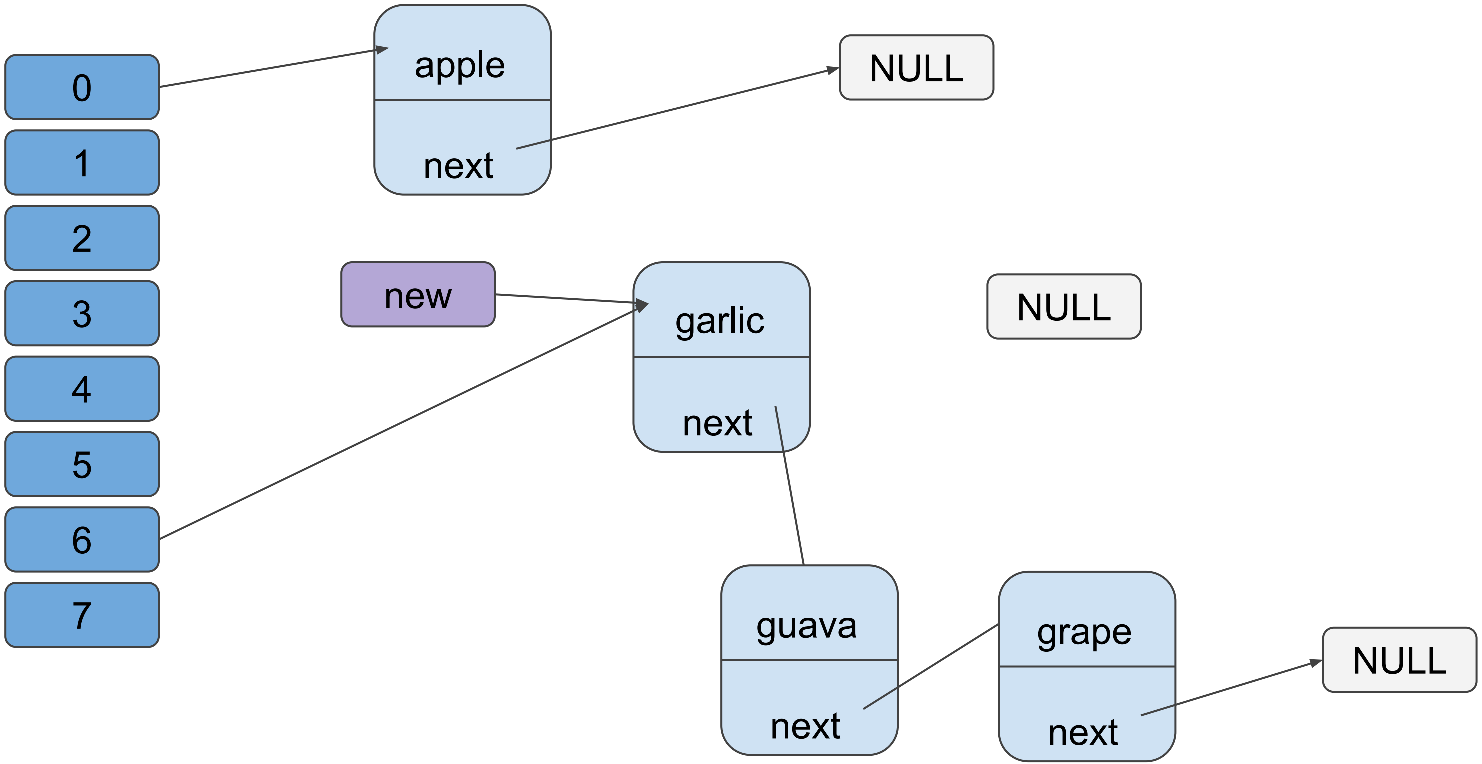
Another solution?

Hash Table: Chaining

- Chaining: each element of the array is a pointer to the head of a linked list, so multiple pieces of data can yield the same hash code and we’ll be able to store it all!

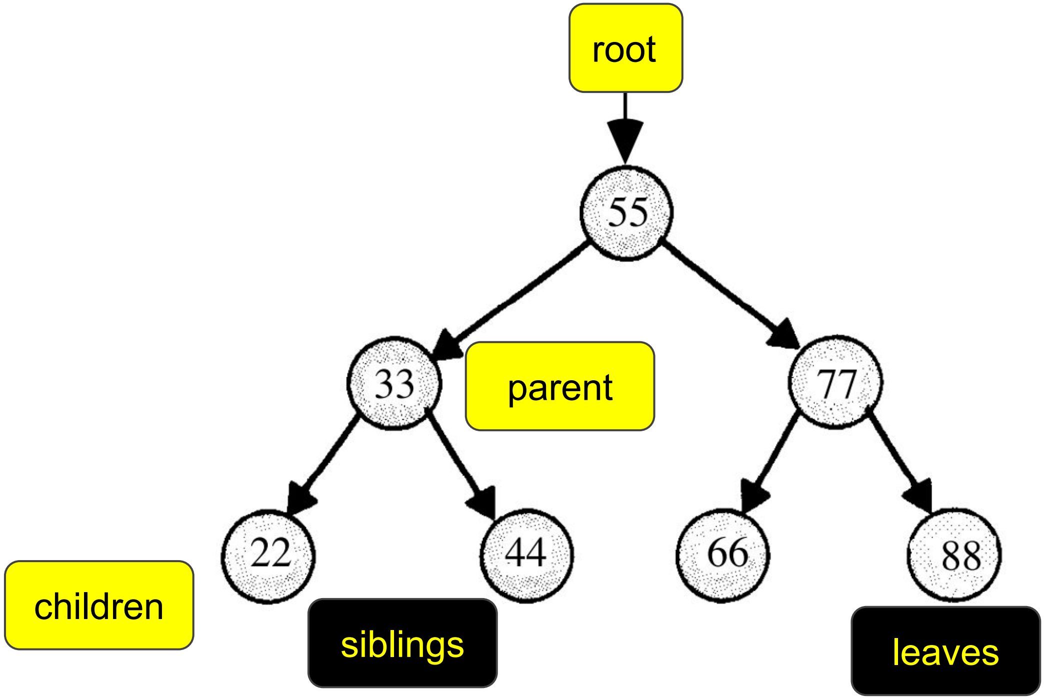


Hash Table: Insert a Node



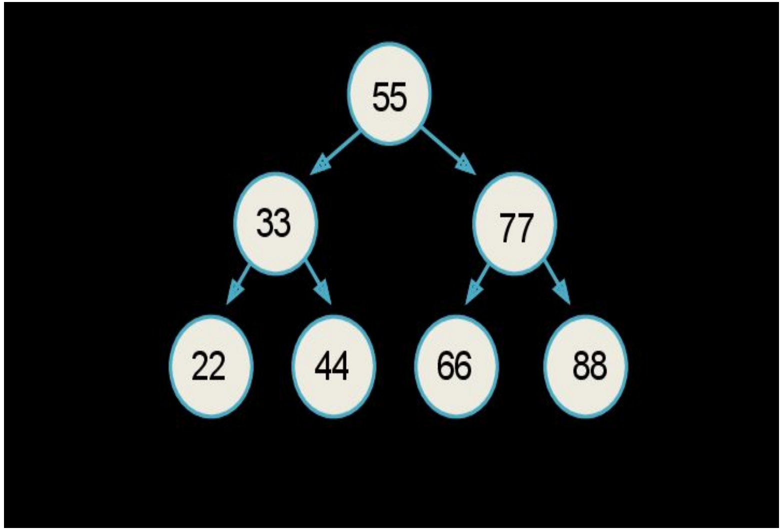
Tree

Any data structure where data is organized hierarchically, with root nodes pointing to child nodes.



Binary Search Tree

- A tree that is organized in such a way as to make binary search more approachable. It is organized such that every parent node can have, at most, 2 child nodes whose positions follow a set pattern, binary search trees greatly simplify data lookup.



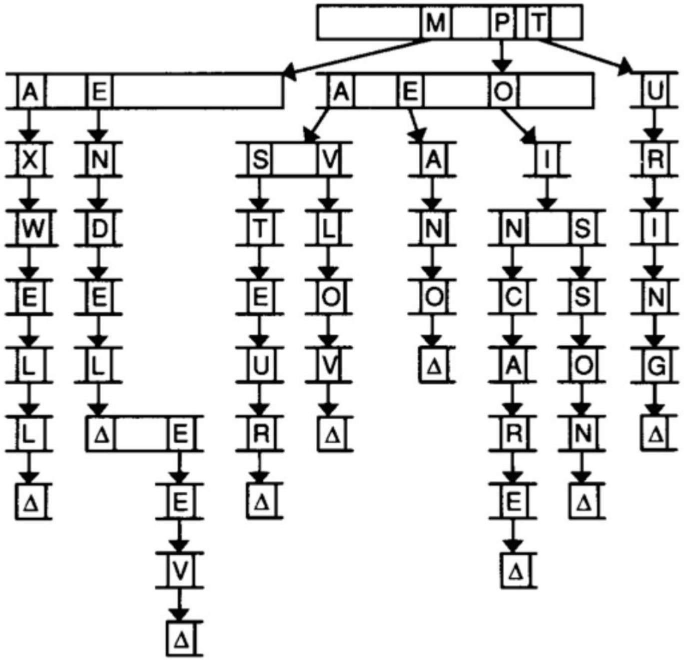
Tree

```c
typedef struct node
{
     int n;
     struct node *left;
     struct node *right;
}
```

```c
bool search(int n, node *tree)
{
     if (tree == NULL)
     {
         return false;
     }
     else if (n < tree->n)
     {
         return search(n, tree->left);
     }
     else if (n > tree->n)
     {
          return search(n, tree->right);
     }
     else
     {
          return true;
     }
}
```

Trie

- A tree with an array as each of its children
- Each array contains pointers to the next layer of arrays.
- In a trie, the data to be searched for is now a roadmap. If you can use the data as a map, and follow the map from beginning to end, then the data must exist in the structure. If you can't, it doesn't.
- No collisions, and no two pieces of data have the same path unless that data happens to already be identical.
- To look for an element, in this case a word, we start with the first letter, then see if the next letter has a child, and continue until we are at the end of our word and see a valid ending.



Trie

```c
typedef struct node
{
    // marker for end of word
    bool is_word;

    // array of node* that
    struct node *children[27];

} node;
```

Data Structure Summary

	Arrays	**Linked lists**	**Hash tables**	**Tries**
insertion	bad	easy	two-step	complex
deletion	bad	deletion	easy	easy
look-up	great	bad	Better than linked list	fast
sort	Relatively easy	Relatively difficult	Not easy	Already sorted
size	Relatively small	Relatively small	Can be big	Can be huge

Stacks and Queues

- Data structures
- can be implemented with either an array or a linked list
- Stacks:
  - LIFO (last in first out)
  - Keep track of size and capacity
  - Push/Pop
- Queues:
  - FIFO (first in first out)
  - Keep track of size, capacity and the start of the queue (aka head)
  - Enque/Deque

Which data structure?

STACKS	QUEUES
	

pset5

Implement a spell checker

- Read the Specification
- Watch this week's [CS50 Shorts]
- CS50 Discourse
- Visit OH