# What do you think?

**Please fill out your course reviews:**

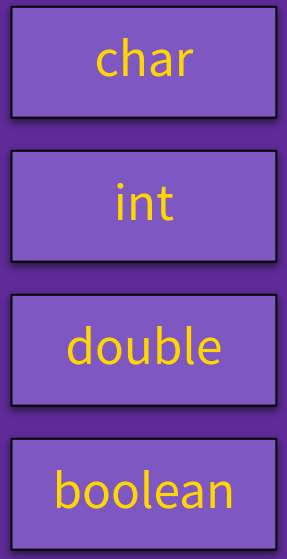
**https://uw.iasystem.org/survey/296999__;!!K-Hz7m0Vt54!nLRI_yn9MCf0zh7Jczxcj2N_Obi60CAXKqf9MepVZkpHu5AXg2y_8bi1zJgMP-0Db9nN20tlzikEXxl5rA$**

# CSE 374: Lecture 29

Data Structures & Decision Trees

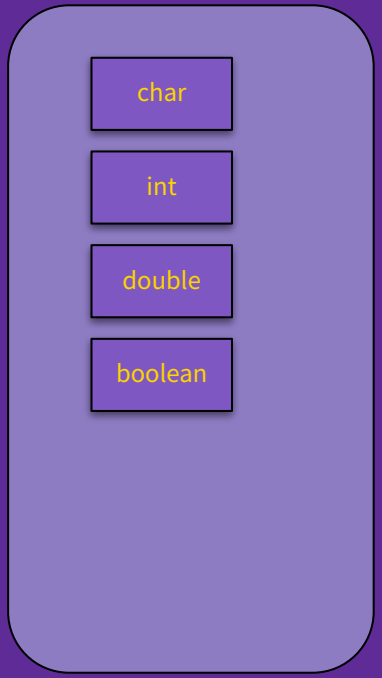
# What is a data structure?

# What is a data structure?



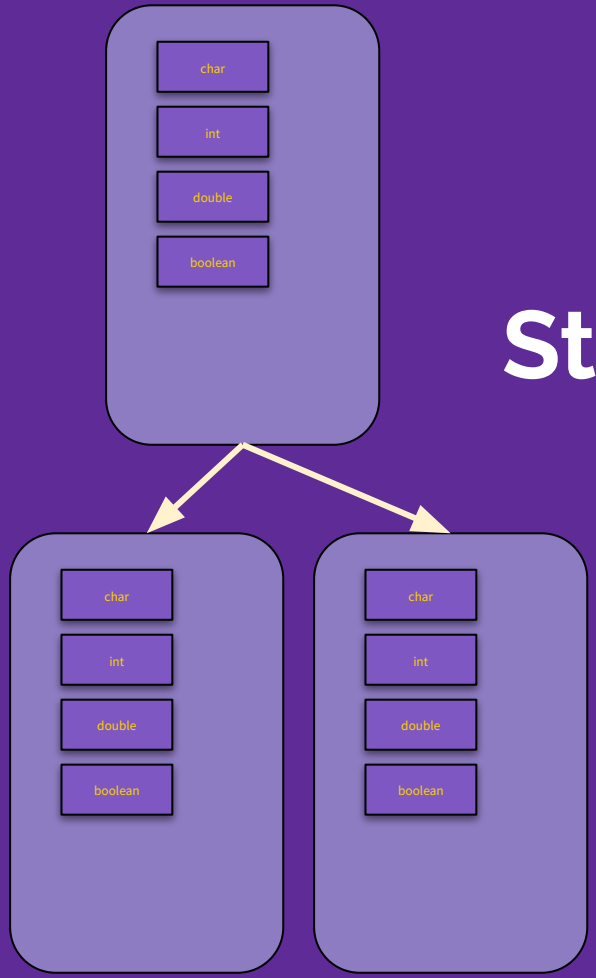

## Primitives

# What is a data structure?




Structs
Classes

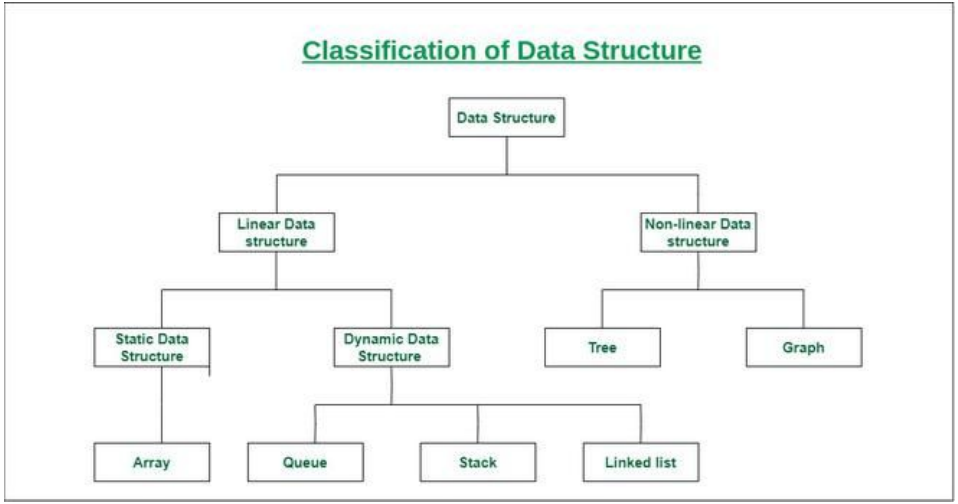
# What is a data structure?




# Data Structures

# Data Structures

- In common:
  - Individual pieces of data are related
  - Organization reflects relationships
- Different:
  - Complexity organization
  - Mutability of organization




Thanks: GeeksforGeeks.org

# Data Structures & Algorithms

- In common:
  - Individual pieces of data are related
  - Organization reflects relationships
- Different:
  - Complexity organization
  - Mutability of organization

Algorithms are intertwined with data structures.

- Structure components inform algorithmic possibilities.
- Structures can be designed to facilitate algorithmic  needs
- Algorithms are designed to react to inherent structures in the data

Thanks: GeeksforGeeks.org

# Data Structures & Algorithms

Consider a list that is only traversed in one direction - single linkage is ok

But doubly linked lists allow for bi-directional movement

Algorithms are intertwined with data structures.

- Structure components inform algorithmic possibilities.
- Structures can be designed to facilitate algorithmic needs
- Algorithms are designed to react to inherent structures in the data

# Data Structures & Algorithms

Arrays & Linked-lists both store sequences of objects

Arrays great for quick random access

Lists great for insertion to the middle

Algorithms are intertwined with data structures.

- Structure components inform algorithmic possibilities.
- Structures can be designed to facilitate algorithmic needs
- Algorithms are designed to react to inherent structures in the data

# Data Structures & Algorithms

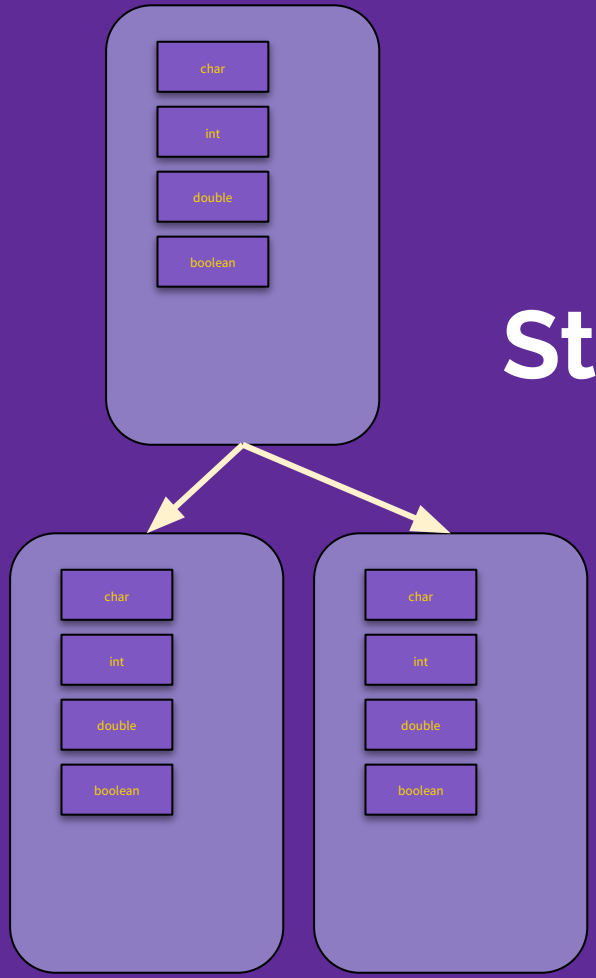
So many tree algorithms!

Because data can frequently be structured in a tree

Algorithms are intertwined with data structures.

- Structure components inform algorithmic possibilities.
- Structures can be designed to facilitate algorithmic  needs
- Algorithms are designed to react to inherent structures in the data

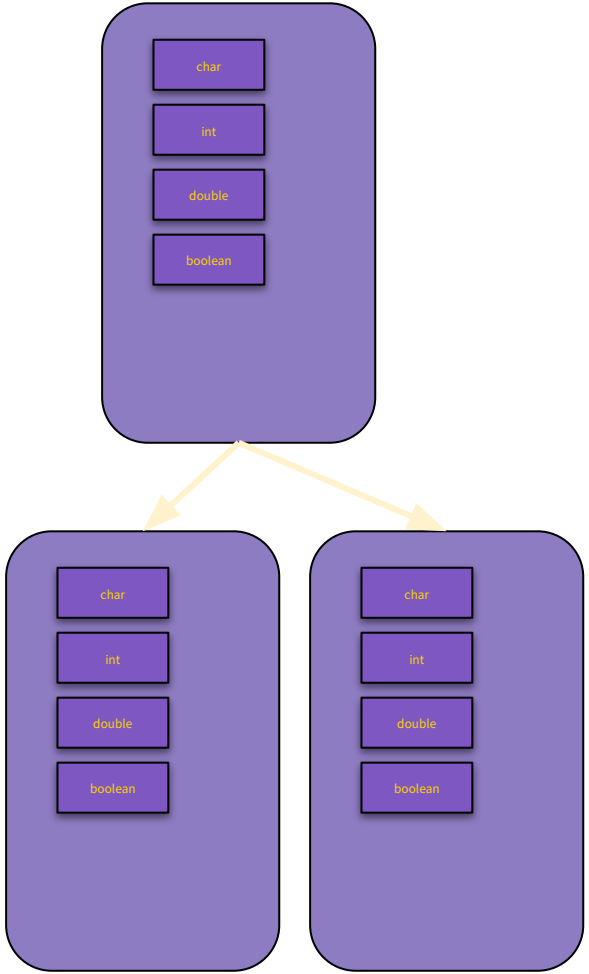
# How do we make these in C?




# Data Structures

# Build up from scratch

- What do you need?
  - What primitives?
  - What structures?
  - What data structures?
    - What nodes are needed?
    - What connections are needed?
- Consider
  - Do you just read the structure
  - Or need to mutate it?
  - How do you move from one object to another?



# Basic Node in C

```
typedef struct Node {

  <primitive-type> data;

  <struct-type> more_data;

  struct Node* next;

  struct Node* branches[B];

} Node;
```

# Basic Node in C

```
typedef struct Node {
  <primitive-type> data;
  <struct-type> more_data;
  struct Node* next;
  struct Node* branches[B];
} Node;
```

What data is needed?

What connections are needed?

# Basic Node in C

```
typedef struct Node {
  <primitive-type> data;
  <struct-type> more_data;
  struct Node* next;
  struct Node* branches[B];
} Node;
```

# Supporting Functions

```
Node* make_node

void* free_node

…connect_nodes …

…traverse_nodes …

…search_nodes …

…print_nodes …
```

# Basic Node in C

**Why do we use pointers here?**

**Does it matter if we stack-allocate or heap-allocate?**

**What does this mean we have to consider?**

# Supporting Functions

```
Node* make_node

void* free_node

…connect_nodes …

…traverse_nodes …

…search_nodes …

…print_nodes …
```

# Basic Node in C++

```
class Node {

private:

  <primitive-type> data;

  <struct-type> more_data;

  Node* next;

  Node* branches[B];

};
```

# Basic Node in C++

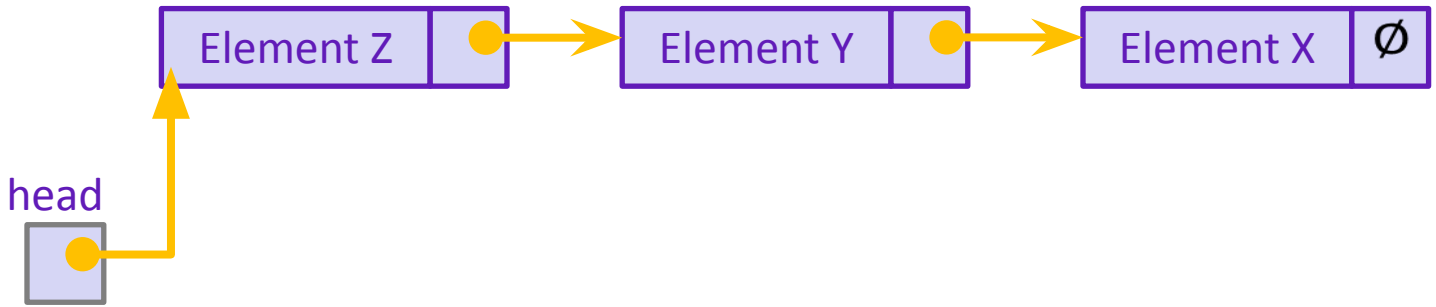
```
class Node {

private:

  <primitive-type> data;

  <struct-type> more_data;

  Node* next;

  Node* branches[B];
```

# Supporting Functions

```
public:

  Node();

  ~Node();

  void addNode (Node*);

  void deleteStructure();

  friend ostream & operator<<
         (ostream &out, const
          Vector &v)
};
```

# Simple Linked List in C

- Each node in a linear, singly-linked list contains:
  - Some element as its payload
  - A pointer to the next node in the linked list
    - This pointer is NULL (or some other indicator) in the last node in the list



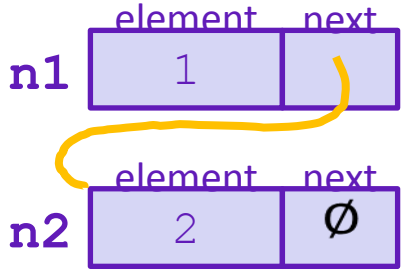
# Linked List Node

- Let's represent a linked list node with a struct
  - For now, assume each element is an `int`

```c
#include <stdio.h>

typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

int main(int argc, char** argv) {
  Node n1, n2;

  n1.element = 1;
  n1.next = &n2;
  n2.element = 2;
  n2.next = NULL;
  return EXIT_SUCCESS;
}
```

manual_list.c

Need to use `struct node_st` here. `Node` not defined until after end of `typedef`.



# Push Onto List

Arrow points to *next* instruction.

```
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

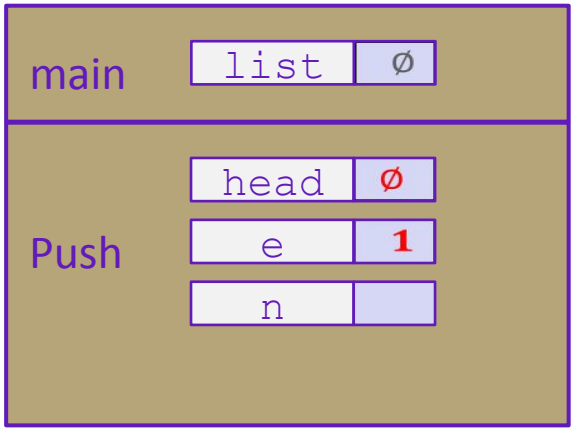

push_list.c

# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

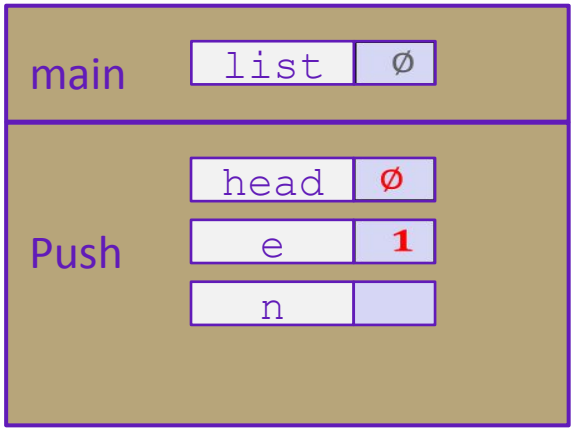


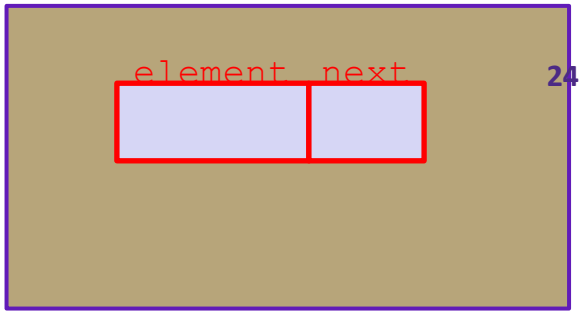
# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c



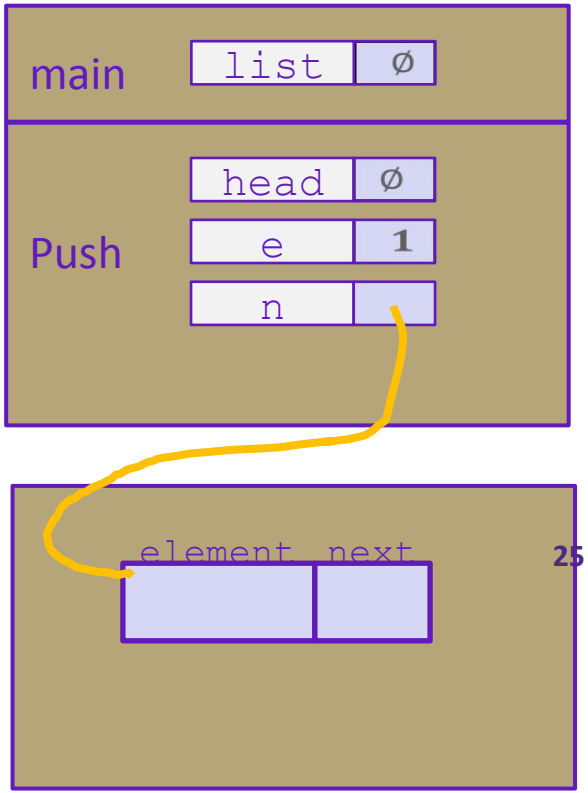



# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

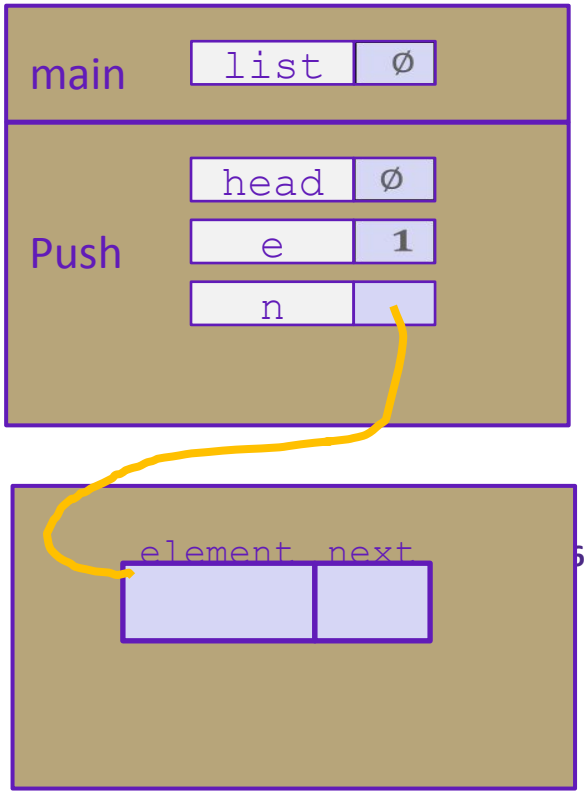


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

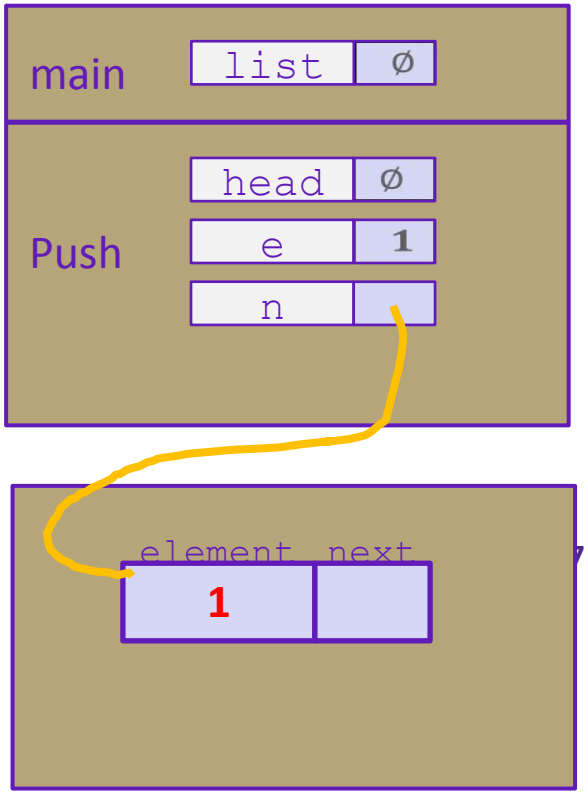


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

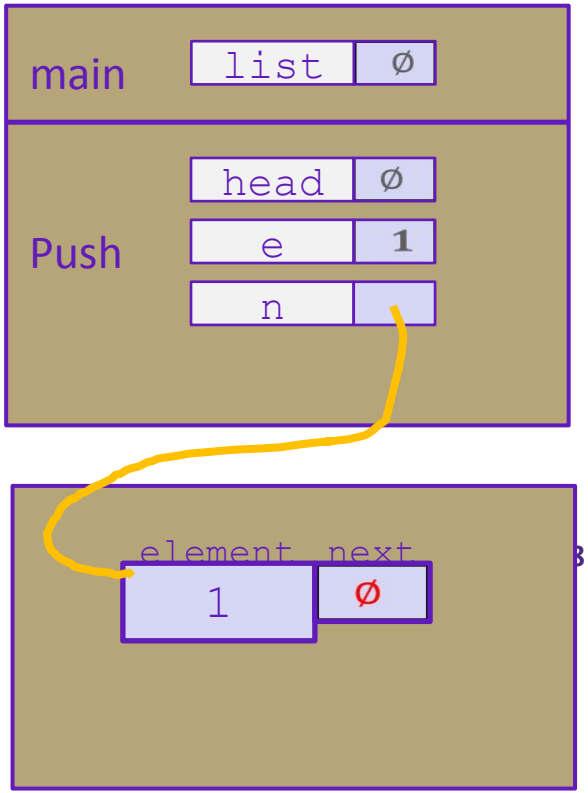


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

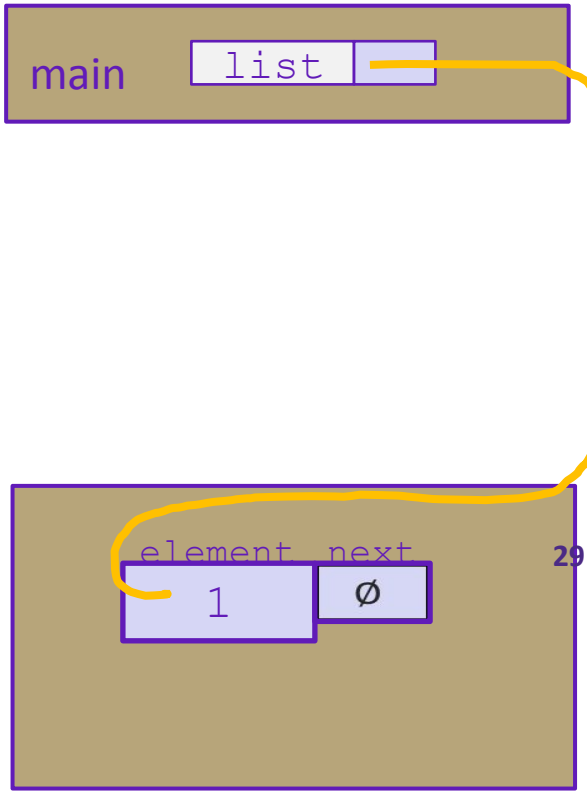


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

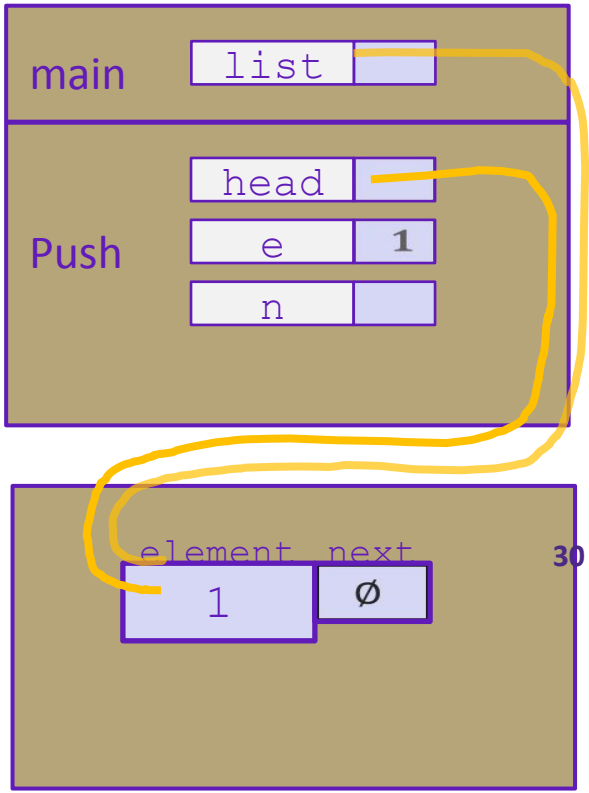


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

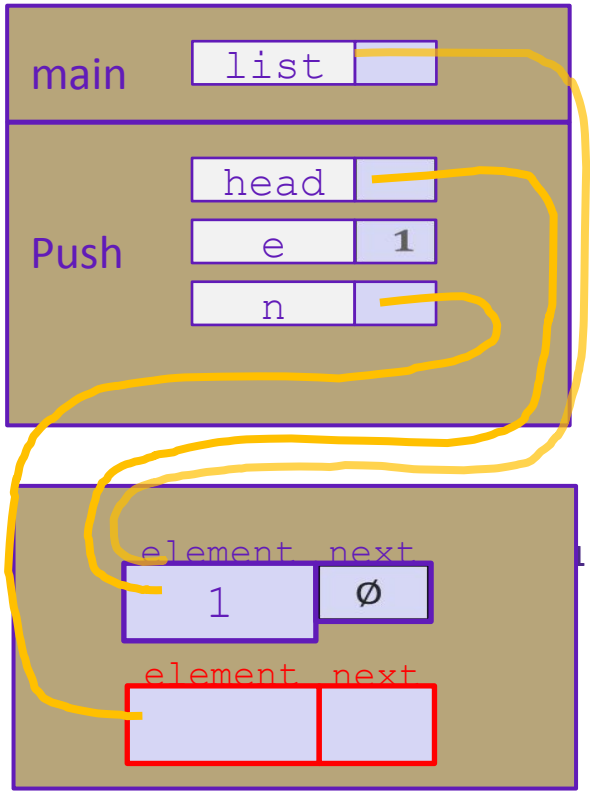


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

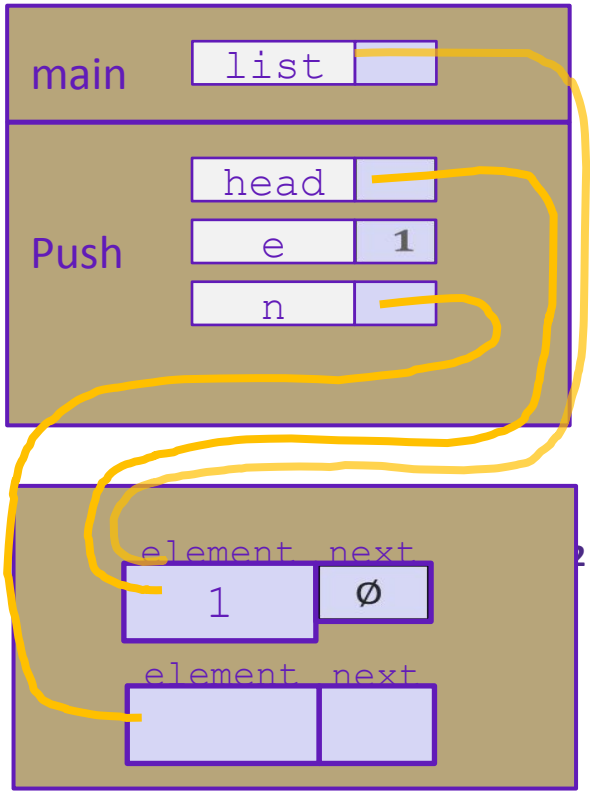


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

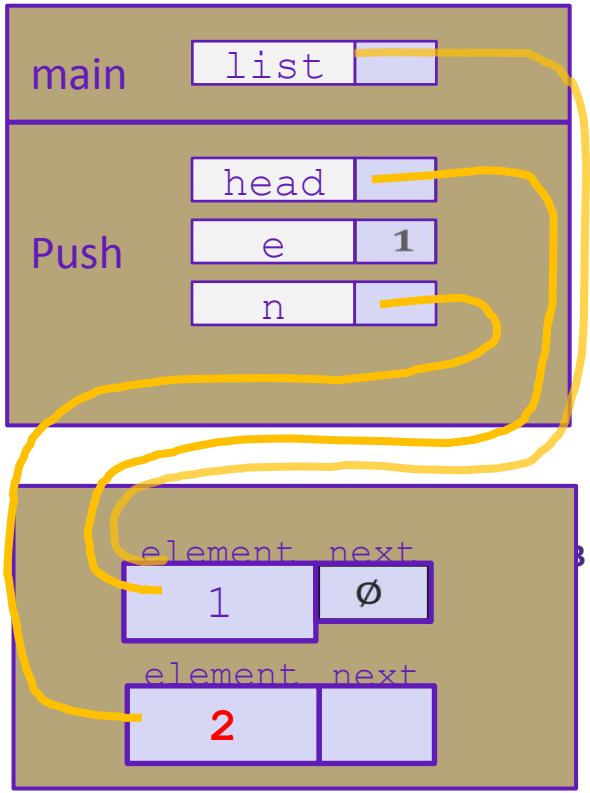


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

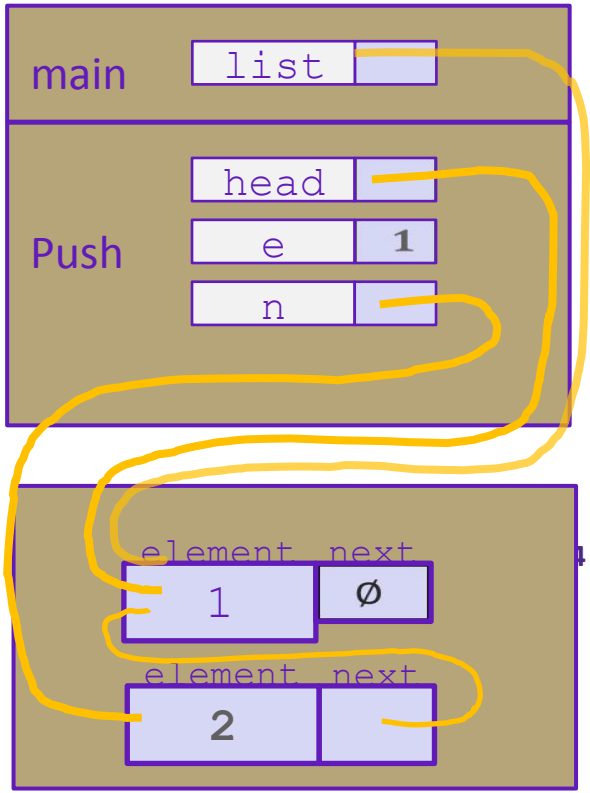


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

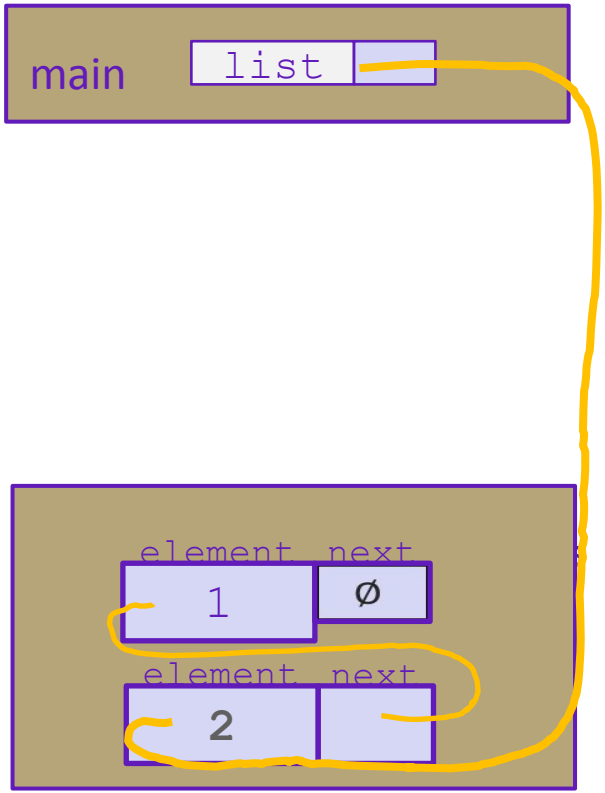


# Push Onto List

Arrow points to *next* instruction.

```c
typedef struct node_st {
  int element;
  struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c



# Decision Trees

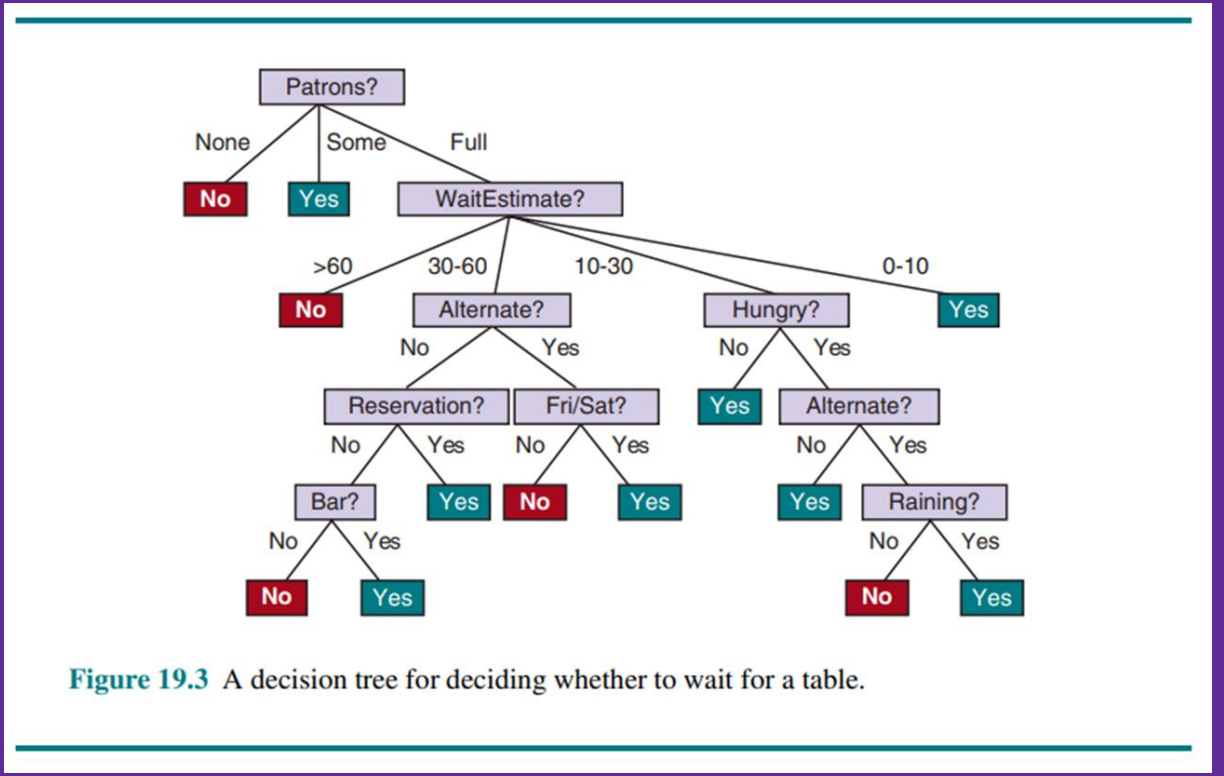
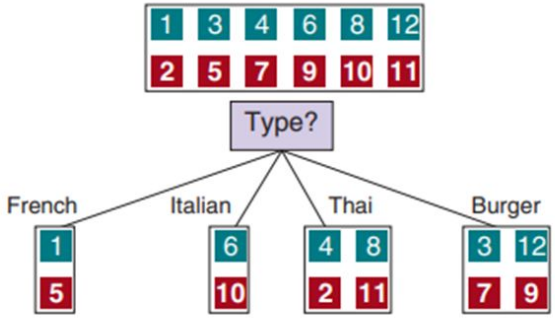
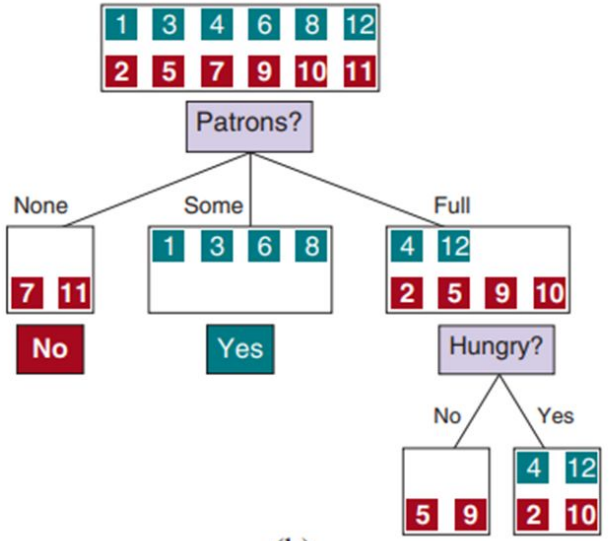



**Figure 19.3** A decision tree for deciding whether to wait for a table.

With thanks to http://ai.berkeley.edu
Figures from Artificial Intelligence: A Modern Approach, Russell & Norvig

# How?




(a) (b)

**Figure 19.4** Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

With thanks to http://ai.berkeley.edu
Figures from Artificial Intelligence: A Modern Approach, Russell & Norvig

# Decision Tree Algorithm

**function** LEARN-DECISION-TREE($examples$, $attributes$, $parent\_examples$) **returns** a tree

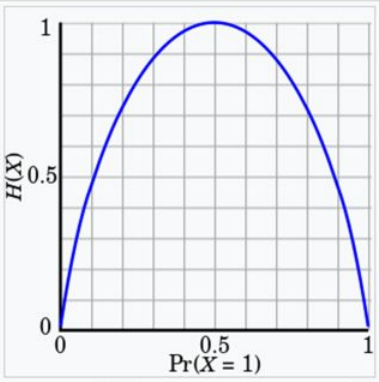
**if** $examples$ is empty **then return** PLURALITY-VALUE($parent\_examples$)
**else if** all $examples$ have the same classification **then return** the classification
**else if** $attributes$ is empty **then return** PLURALITY-VALUE($examples$)
**else**
  $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE($a$, $examples$)
  $tree \leftarrow$ a new decision tree with root test $A$
  **for each** value $v$ of $A$ **do**
    $exs \leftarrow \{e : e \in examples$ **and** $e.A = v\}$
    $subtree \leftarrow$ LEARN-DECISION-TREE($exs$, $attributes - A$, $examples$)
    add a branch to $tree$ with label $(A = v)$ and subtree $subtree$
  **return** $tree$

**Figure 19.5** The decision tree learning algorithm. The function IMPORTANCE is described in Section **??**. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

Russell & Norvig

# Importance

- $H(X) = -\sum_{x \in X} p(x) \log p(x)$
  - Measures ‘information’ or ‘surprise’ in data
- Information: $IG(A) = H - \sum_{v=A} \frac{H_v}{H} H_v$
  - Measures ‘reduction in entropy’ given a decision on *f*
- *(Other methods)*
  - *Entropy has continuous data form*
  - *Chi-square*
  - *Etc.*



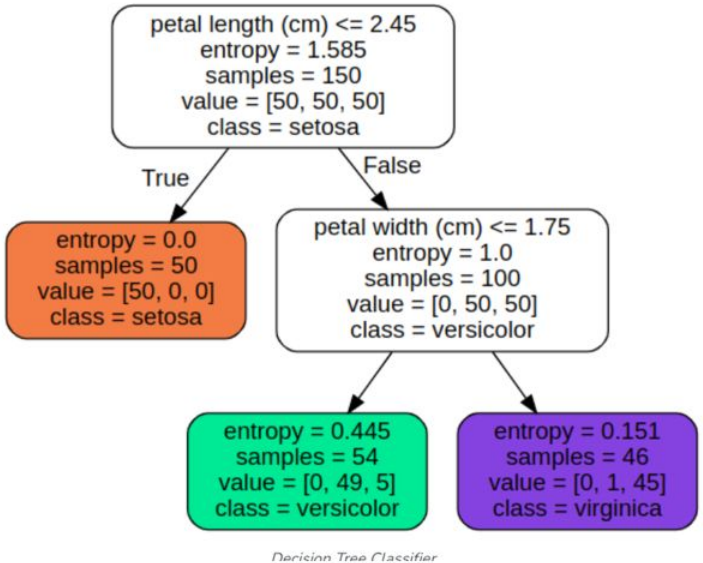

Entropy $\mathrm{H}(X)$ (i.e. the expected surprisal) of a coin flip, measured in bits, graphed versus the bias of the coin $\Pr(X = 1)$, where $X = 1$ represents a result of heads.[10]: 14–15

Here, the entropy is at most 1 bit, and to communicate the outcome of a coin flip (2 possible values) will require an average of at most 1 bit (exactly 1 bit for a fair coin). The result of a fair die (6 possible values) would have entropy $\log_2 6$ bits.

wikipedia

# Decision Trees: pros and cons

- Straight forward
- Can be used for regression as well as classification
- Divisions are on features, explicitly.
  - That means we can track why something got put into a certain category
  - Which side of the feature divide was it on
- Easy to over train
- Not great with non-linear data




Geeks for Geeks

# Decision Tree Nodes:

```
20 class DecisionNode {
21   friend class DecisionTree;
22 public:
23   DecisionNode(const int dec_ind, const double dec_cut);
24   DecisionNode(const int dec_ind, const double dec_cut, const string label);
25   ~DecisionNode();
26
27   void setBelow (DecisionNode* b) {below_ = b;};
28   void setAbove (DecisionNode* a) {above_ = a;};
29
30   friend void delete_tree (DecisionNode* root);
31   friend void print_tree (DecisionNode* root);
32   friend ostream & operator<<(ostream &out, const DecisionNode &n);
33
34 private:
35   int decision_index_;     // highest information feature index
36   double decision_cutoff_; // dividing line
37   string *label_;          // label if leaf node
38   DecisionNode* below_;
39   DecisionNode* above_;
40 };
```

# Decision Tree Structure:

```
class DecisionTree {
  friend class DecisionNode;

public:
  DecisionTree(int num_feat, string* &feat_names,
               int num_lab, string* &lab_names);
  ~DecisionTree();

  // We will assume that the data set dimensions are correct for now.
  // it would be better to cross check the # of features and samples
  void train (const int num_samples, double** data, string* labels);
  double test (const int num_samples, double** data, string* labels);
  string get_label (double* sample);
  void print ();

private:
  int num_features_;
  string* feature_names_;
  int num_labels_;
  string* label_names_;
  DecisionNode* root_;

  // used internally to run recursive training algorithm
  DecisionNode* train_node (const int num_samp, double** data,
                            const string* labels);
  void delete_tree (DecisionNode* root);
};
```

# Decision Tree Implementation:

```
DecisionNode::DecisionNode (const int dec_ind, const double dec_cut) :
  decision_index_(dec_ind), decision_cutoff_(dec_cut),
  above_(nullptr), below_(nullptr), label_(nullptr) {
}

DecisionNode::DecisionNode(const int dec_ind, const double dec_cut,
                           const string label) :
  decision_index_(dec_ind), decision_cutoff_(dec_cut),
  above_(nullptr), below_(nullptr) {
  label_ = new string(label);
}

DecisionNode::~DecisionNode() {
  delete label_;
  delete above_;
  delete below_;
}
```

Constructors:

- Use the initializer syntax to initialize values
- Set unused pointers to nullptr
- Use constructor to allocate space for the string and copy a value into it.

Destructor:

- Need to explicitly delete any allocated memory (the label and the two links).
- Nicely, using the destructors, this recursively deletes children nodes so the entire tree is deleted.