What do you think?

Work together!

```
typedef struct trieNode {
    ______ _branches_____;
    char *word;
} trieNode;
```

Fill in the blanks to declare a struct that allows for an array of branches. Each branch is a pointer to a new trieNode.

CSE 374 Lecture 14

More Data Structures

Structs Reminder

Has type struct person_info

‘Person_info’ is a struct tag, not a type

Can use typedef to rename

```
typdef struct person {
   char * name;
   int age;
} person;
```

Structs Reminder

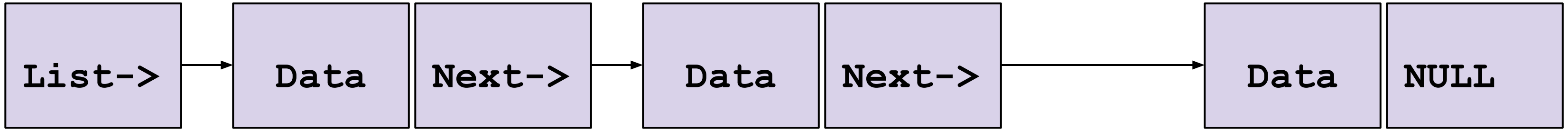
Has type struct person_info

‘Person_info’ is a struct tag, not a type

Can use typedef to rename

```
struct person {
    char * name;
    int age;
}
typedef struct person person;
```

Linked Lists



Points to
the List

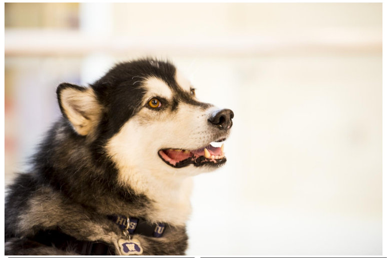
Last node doesn’t
point to next

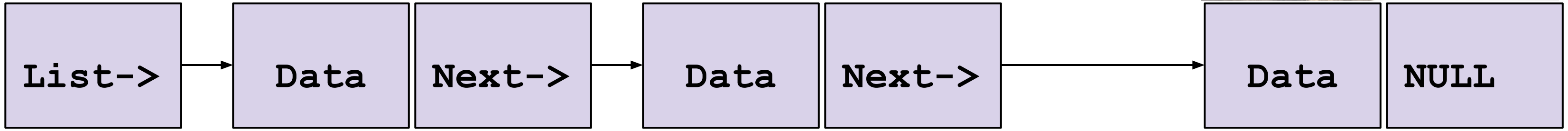
```
// A single list node that stores an
integer as data.

typdef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;
```

```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  if (n) {  // malloc might return null
    n->data = data;
    n->next = next;
  }
  return n;
}
```

Linked Lists





Points to
the List

Last node doesn't
point to next

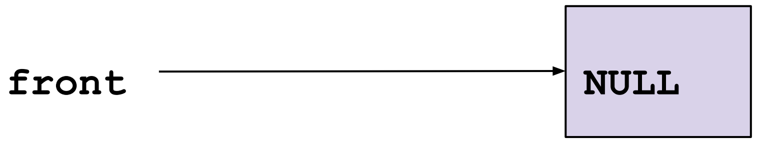
```
// A single list node that stores an
integer as data.

typdef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;
```

```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode n;
  n->data = data;
  n->next = next;
  return &n;
}
```

Why not this?

Linked Lists

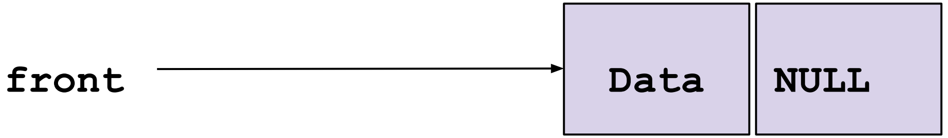


```
IntListNode* fromArray(int* array, int length) {
  IntListNode* front = NULL;
  for (int i = length - 1; i >= 0; i--) {
    front = makeNode(array[i], front);  }
  return front;
}
```

```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  if (n) {  // malloc might return null
    n->data = data;
    n->next = next;
  }
  return n;
}
```

Linked Lists

```
IntListNode* fromArray(int* array, int length) {
  IntListNode* front = NULL;
  for (int i = length - 1; i >= 0; i--) {
    front = makeNode(array[length-1], front);  }
  return front;
}
```



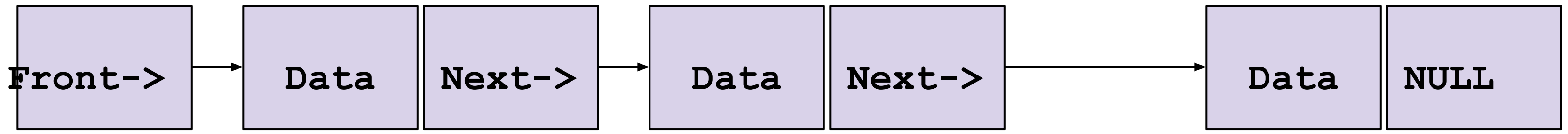
Last node doesn’t point to next

```
// A single list node that stores an
integer as data.

typdef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;
```

```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  if (n) {  // malloc might return null
    n->data = data;
    n->next = next;
  }
  return n;
}
```

Linked Lists



Points to
the List

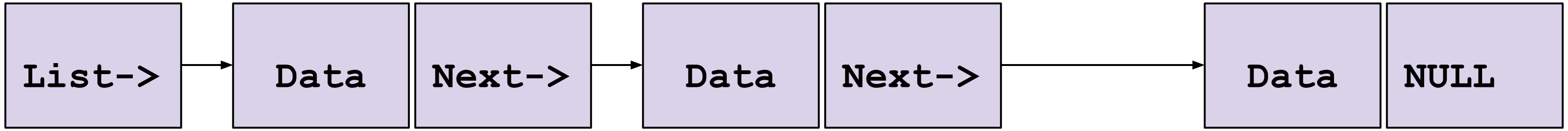
Last node doesn’t
point to next

```
// A single list node that stores an
integer as data.

typdef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;
```

```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  if (n) {  // malloc might return null
    n->data = data;
    n->next = next;
  }
  return n;
}
```

Linked Lists



Points to
the List

Last node doesn’t
point to next

```
// A single list node that stores an
integer as data.

typdef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;
```

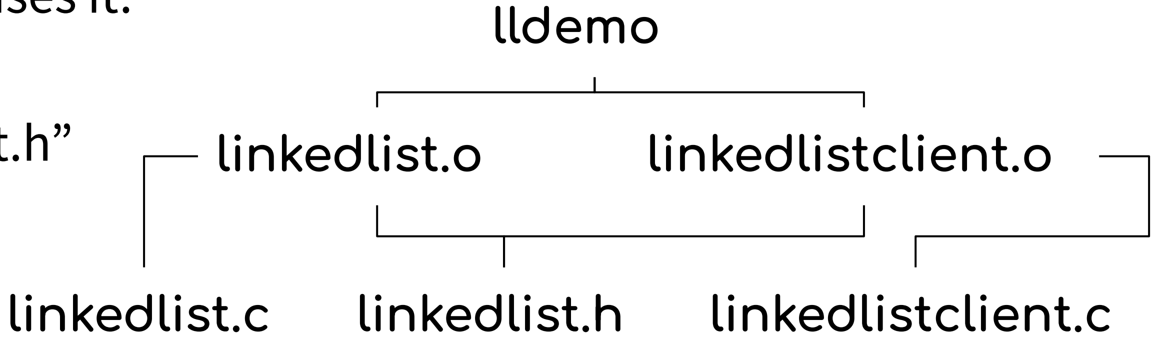
```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  if (n) {  // malloc might return null
    n->data = data;
    n->next = next;
  }
  return n;
}
```

Linked List Continued

- One set of code to define linked list:
  - Linkedlist.h
  - Linkedlist.c
- Another piece of code uses it:
  - Linkedlistclient.c
  - #include “linkedlist.h”

Compile with

```
$gcc -o lldemo linkedlist.c
    linkedlistclient.c
```



Memory management and data structures.

```
void freeList(IntListNode* list) {
  while (list != NULL) {
    IntListNode* next = list->next;
    free(list);
    list = next;
  }
}

void printList(IntListNode* list) {
  printf("[");
  while (list != NULL) {
    printf(" %d", list->data);
    list = list->next;
  }
  printf(" ]\n");
}
```

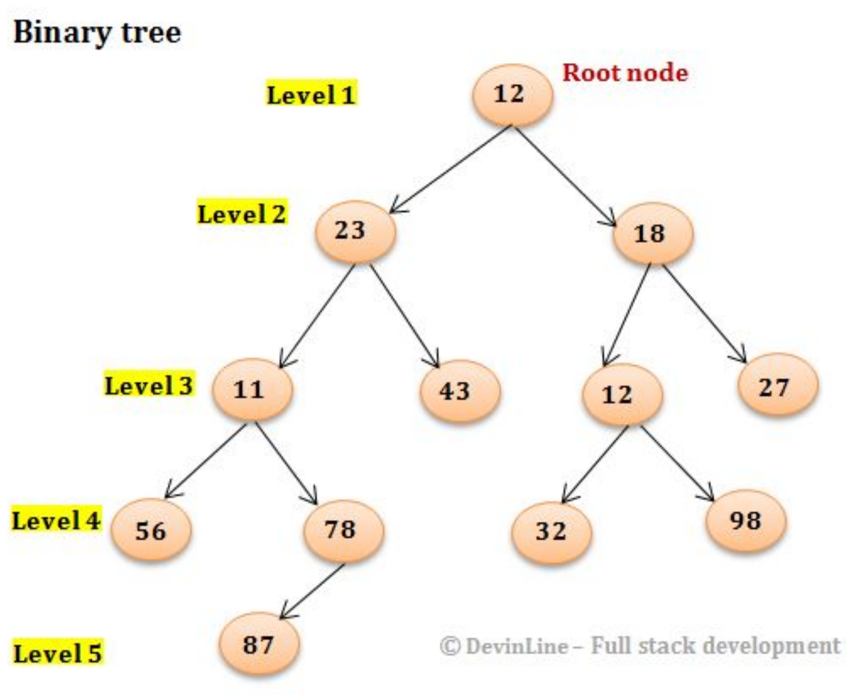
Notice that the makeNode function dynamically allocates memory, and then returns a pointer to that memory.

We must free that memory. Usually this will be another function that frees a list.

You need to be careful not to lose the pointer to front.

Luckily, if you pass that pointer as an argument, the value of the address is copied over. You can use this access the memory at that location, and you can change the copy local to the function without changing the original copy.

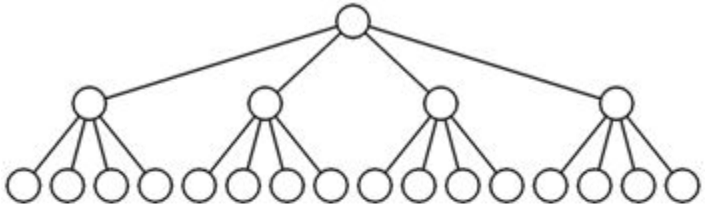
Binary Trees



```
struct BinaryTreeNode {
  int data;
  struct BinaryTreeNode* left;
  struct BinaryTreeNode* right;
}

struct BinaryTree {
  Struct BinaryTreeNode* root;
}
```

N-ary Trees



```
struct TrinaryTreeNode {
  char* data;
  struct TrinaryTreeNode* left;
  struct TrinaryTreeNode* middle;
  struct TrinaryTreeNode* right;
}
```

```
struct QuadTreeNode {
  char* data;
  struct QuadTreeNode* children[4];
}
```

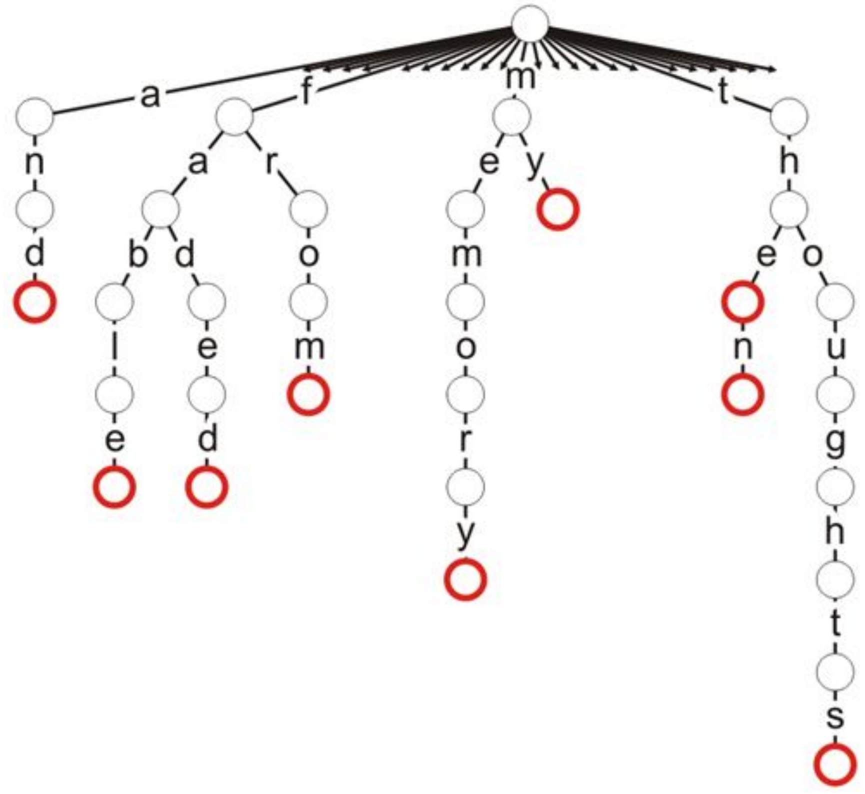
Binary trees just one form; can have any “branching number”.

Trinary trees have branching number of three.

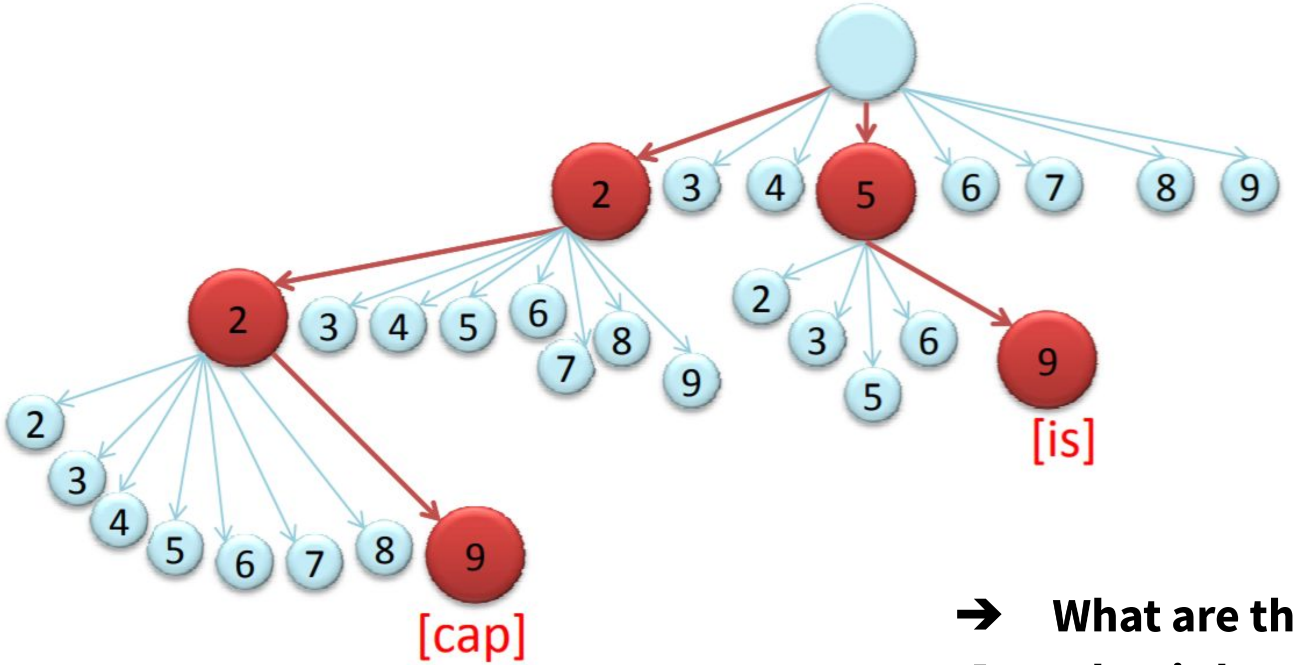
For arbitrarily large branching numbers, arrays can make more sense than lists of named pointers.

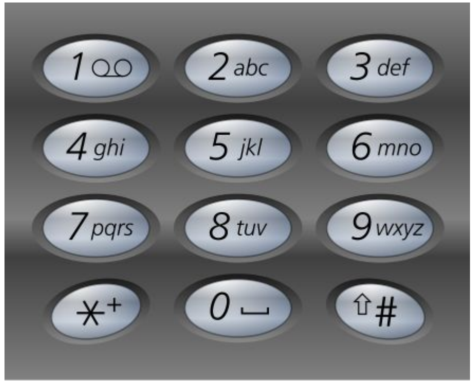
Prefix tree (Trie)

- Compact storage
  - Or generative automaton
- Key of each node defined entirely by position
- Compact data storage
- Efficient worst-case searching
- Strings often use 26-ary tree
  - Predictive text
  - Spell check



T9 Trie





- ➔ **What are the branches labeled?**
- ➔ **What is branching factor?**
- ➔ **What data is stored in each node?**