{C}

Testing

Testing

A process of evaluating a particular product to determine whether the product contain any defects.

Devil's Advocate:

- *"Program testing can be used to show the presence of defects, but never their absence".* Dijkstra

- In defence:
- What is Correctness anyway?
- Builds Confidence.

Correctness

Program Correctness

- A program P is considered with respect to a specification S, if and only if:

  For each valid input, the output of P is in accordance with the specification S

What if the specifications are themselves incorrect?

Levels of testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing

The methodology

The derivation of test inputs is based on *program specifications*.

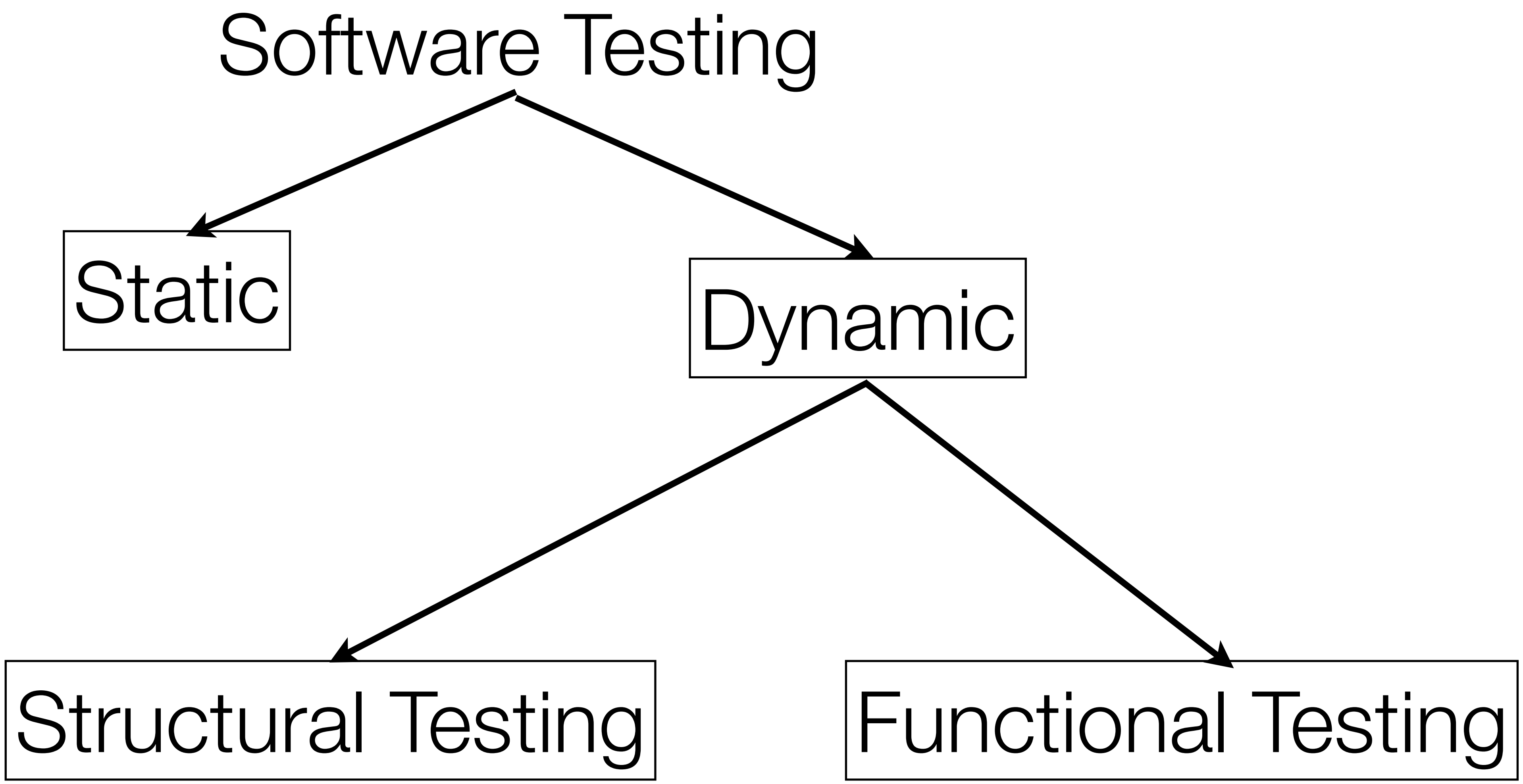
Clues are obtained from the specifications.

Clues lead to *test requirements*.

Test requirements lead to *test specifications*.

Test specifications are then used to actually execute the program under test.

Software Testing



Software Testing

Static

Dynamic

Structural Testing

Functional Testing

Linked List Testing

How do we apply this to Linked Lists?

- It is just a list..There is nothing we can do about it!

Linked List Testing

How do we apply this to Linked Lists?

- It is just a list..There is nothing we can do about it!

WRONG!

Linked List functionality

```
struct lnode {
  struct lnode* prev;
  struct lnode* next;
  char* word;
  int count;
  int line;
};
struct lnode *nodeGetNext(struct lnode *);
struct lnode *newNode(struct lnode **, struct lnode **, char *, int);
struct lnode *getNode(struct lnode **, struct lnode **, char *, int);
struct deleteList(struct lnode **, struct lnode **);
void deleteNode(struct lnode **, struct lnode **, struct lnode *);
char *nodeGetWord(struct lnode *);

struct **lnode head;
struct **lnode tail;
```

Start Early

Write your test cases and make sure that they fail before you start implementing the linked list functionality.

Test even before implementing.

Supporting tests

```
/*create a struct to handle our test data*/
struct listTest {
  /*count the number of nodes in a list*/
  int count;
  void (*verifyCount)(struct lnode*);
};

struct listTest* myListTest;
/*checks that the list actually has the correct number of nodes*/
void assert_ListSize(struct lnode* listHead){
  int counter = 0;
  struct lnode* temp = listHead;
  while(temp!=NULL){
    temp=temp->next;
    counter++;
  }
  if(counter!=myListTest->count){
    fprintf(stderr,”Error (1) Size do not match”);
  }
}
```

Supporting tests

```
/*create a struct to handle our test data*/
struct listTest {
  /*count the number of nodes in a list*/
  int count;
  void (*upCount)(int);
  void *verifyCount(struct lnode*);
};
struct listTest* myListTest;
/*update test count count*/
void maintain_updateCount(int delta){
  myListTest->count+=delta;
}
```

Supporting tests - continue

```
/*create a struct to handle our test data*/
struct listTest {
  /*count the number of nodes in a list*/
  int count;
  void (*upCount)(int);
  void *verifyCount(struct lnode*);
  void *verifyNULL(struct lnode*);
};
/* checks that the node is not NULL */
/*
 * We can use this assertion to make sure that head/tail is not
 * equal to null after any operation. Unless the count is zero
 */
void assert_NodeNotNull(struct lnode* node){
  if(node==NULL){
    fprintf(stderr,”Error (2) Node is Null\n”);
    exit(1);
  }
}
```

Supporting tests - continue

Before you start coding your linked list, all test cases should fail.

Example, test Insertion:

```
newNode(head, tail, word, line);
/*
 * Because we have empty implementation in newNode, the following
 * test should fail.
 */
myListTest->verifyNULL(head); //Failure
```

Supporting tests - continue

We should start writing our code now:

```
struct lnode *newNode(struct lnode **head, struct lnode **tail, char
*word, int line) {
 /*allocate memory for node*/
 struct lnode * node = (struct lnode *) malloc(sizeof(struct
 lnode));
 if (*head == NULL) {
   *head = node;
   *tail = node;
 }
 else{
  //Later
 }
}
```

Usage:

```
newNode(head, tail, &c, 1);
myListTest->verifyNULL(*head); //success
myListTest->verifyNULL(*tail); //success
myListTest->verifyCount(*head); //failure
```

Supporting tests - continue

```
struct lnode *newNode(struct lnode **head, struct lnode **tail, char
*word, int line) {
  /*allocate memory for node*/
  struct lnode * node = (struct lnode *) malloc(sizeof(struct
  lnode));
  /*
   * Inform the test counter that we added a new node */
   * the count test case should succeed.
   */
  myListTest->upCount(1);
  if (*head == NULL) {
      *head = node;
      *tail = node;
  }
  else{
    //Later
  }
}
```

Supporting tests - continue

We did not make any sanity checks for the node itself.

```
/*create a struct to handle our test data*/
struct listTest {
  /*count the number of nodes in a list*/
  int count;
  void (*upCount)(int);
  void (*verifyCount)(struct lnode*);
  void (*verifyNULL)(struct lnode*);
  void (*verifyNodeData)(struct lnode*, struct lnode*);
};
/* checks that the node data is correct*/
/*
 * We can use this to make sure that we initialize
 * the data correctly
 */
void assert_nodeData(struct lnode* refNode, struct lnode* node){
  if(node->count != refNode->count  ||
       node->line != refNode->line ||
       node->next != refNode->next ||
       node->prev != refNode->prev ||
       (strcmp(node->word, refNode->word)!=0))
    fprintf(stderr,"Error (3) Data Incorrect\n");
```

Supporting tests - continue

Fix our code again to fix the data..

```
struct lnode *newNode(struct lnode **head, struct lnode **tail, char
*word, int line) {
  /*allocate memory for node*/
  struct lnode* node = (struct lnode*) malloc(sizeof(struct lnode));
  myListTest->upCount(1);
  if(*head == NULL) {
      *head = node;
      *tail = node;
  }
  else{
    //Later
  }
  /*The next block should fix the data*/
  node->count=1;
  node->line=line;
  node->word=(char *)malloc(sizeof(char)*strlen(word));
  strcpy(node->word,word);
  node->next=NULL;
  node->prev=NULL;
}
```

Supporting tests - continue

```
struct lnode* checkNode = newNode(head, tail, &c, 1);
struct lnode* ref_node = newNode(head, tail, &c, 1);
ref_node->line=1;
ref_node->next=NULL;
ref_node->prev=NULL;
ref_node->word=&c;

/** test we initialized the correct values*/
myListTest->verifyNULL(*head); //success
myListTest->verifyCount(*head); //success
myListTest->verifyNULL(*tail); //success
myListTest->verifyNodeData(checkNode, ref_node); //success
```

Supporting tests - continue

We do not create the linked list correctly. We will fail when we insert a second node

```
struct lnode* checkNode = newNode(head, tail, &c, 1);
struct lnode* ref_node = newNode(head, tail, &c, 1);
ref_node->line=1;
ref_node->next=NULL;
ref_node->prev=checkNode;
ref_node->word=&c;

struct lnode* secondNode = newNode(head, tail, &c, 1);
myListTest->verifyNULL(*head); //success
myListTest->verifyCount(*head); //failure
myListTest->verifyNULL(*tail); //success
myListTest->verifyNodeData(secondNode, ref_node); //failure
```

Supporting tests - continue

```
struct lnode *newNode(struct lnode **head, struct lnode **tail, char
*word, int line) {
  struct lnode * node = (struct lnode *) malloc(sizeof(struct
  lnode));
  myListTest->upCount(1);
  if (*head == NULL) {
      *head = node;
      *tail = node;
      node->next=NULL; node->prev=NULL;
  }
  else{
    node->next=NULL;
    node->prev=(*tail);
    (*tail)->next=node;
    (*tail)=node;
  }
  node->count=1; node->line=line;
  node->word=(char *)malloc(sizeof(char)*strlen(word));
  strcpy(node->word,word);
}
```

Supporting tests - continue

More Complicated Tests:

Build a test case to detect whether your linked list has a cycle.

```
/*
 * We can use this assertion to detect cycles within a linked
 * list. The idea is simply based on having two iterators with
 * one iterating at twice the speed of the other.
 */
void assert_NoCycle(struct lnode* slow, struct lnode* fast){
  while((fast=fast->next)!=NULL){
    if(fast==slow) { fprintf(stderr,”Error (4) Cycle\n”);}
    fast=fast->next;
    if(fast==slow) { fprintf(stderr,”Error (4) Cycle\n”);}
    slow=slow->next;
  }
}
```

Usage:

```
struct lnode* fast=*head;
struct lnode* slow=*head;
assert_NoCycle(slow,fast);
```

Supporting tests - continue

More Complicated Tests:

Tail is correct?

```
/*
 * We can use this assertion to verify that tail is correctly
 * updated. The tail should be at the end of the list. We can use
 * the test count to check this property.
 */
/*checks that the list actually has the correct number of nodes*/
void assert_CorrectTail(struct lnode* listHead, struct lnode*
listTail){
  int counter = 0;
  struct lnode* temp = listHead;
  while(counter<myListTest->count){
    if(temp==listTail&&counter!=myListTest->count-1{
      fprintf(stderr,”Error (5) Tail\n”);
    }
    count++;
    temp=temp->next;
  }
}
```

Same will apply for deleteNode

```
void deleteNode(struct lnode **head, struct lnode **tail, struct
lnode *node) {
  /*implementation goes here*/

  /* We should update the count when we remove an element*/
  myListTest->upCount(-1);
}
```

More Testing

GDB.

Memory tools.