DATA STRUCTURES
CS-UH 1050, SPRING 2020

Lecture V – Linked lists and Recursion

Prof. Mai Oudah

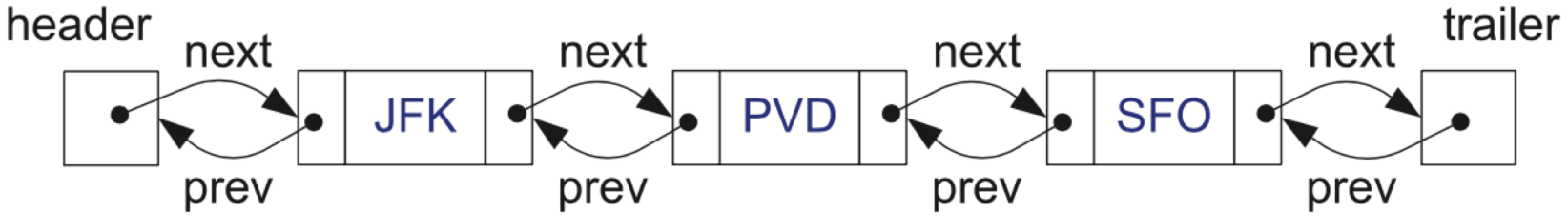
LECTURE OUTLINES

I. Linked lists
II. Recursion

LINKED LISTS

DOUBLY LINKED LISTS

- It allows us to go in both directions—forward and reverse—in a linked list. Such lists allow for quick update operations (insertion/deletion at any point)
- A node in a doubly linked list stores two pointers:
  - *next:* points to the next node in the list
  - *prev:* points to the previous node in the list
- It has two special nodes that do not store any elements:
  - *header* node: just before the head of the list
  - *trailer/tail* node: just after the tail of the list



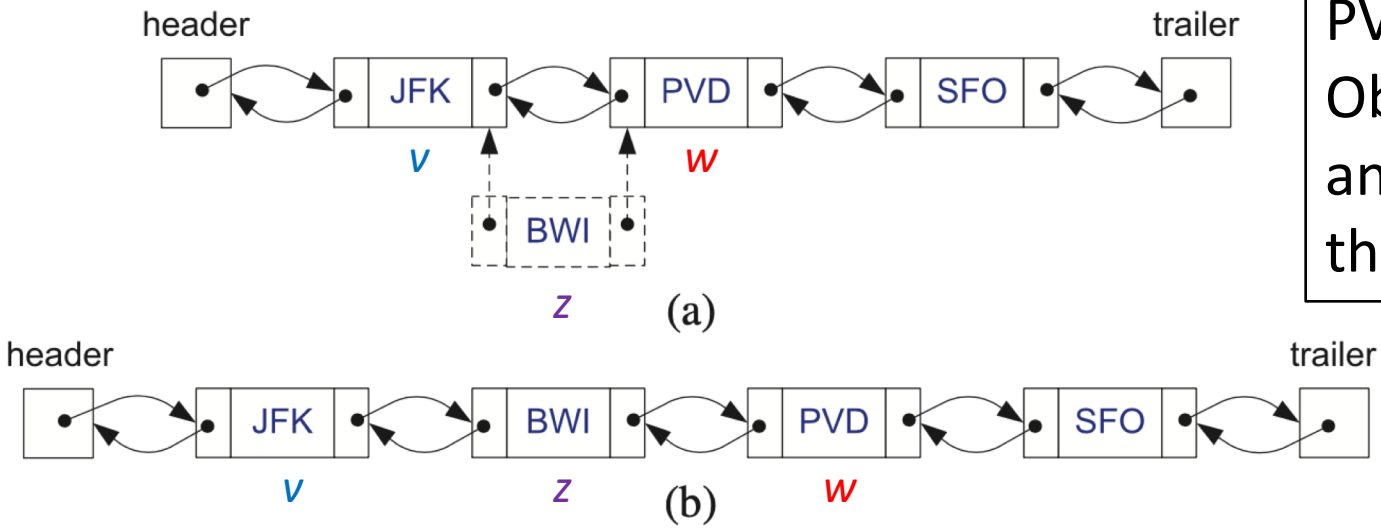
DOUBLY LINKED LIST: INSERTION

- It is possible to insert a node at any position.
- Let *w* be the node after *v*. To insert *z* after *v*, we link it into the current list as follows:

1. Make *z*'s *prev* link point to *v*
2. Make *z*'s *next* link point to *w*
3. Make *w*'s *prev* link point to *z*
4. Make *v*'s *next* link point to *z*

DOUBLY LINKED LIST: INSERTION

- Example:

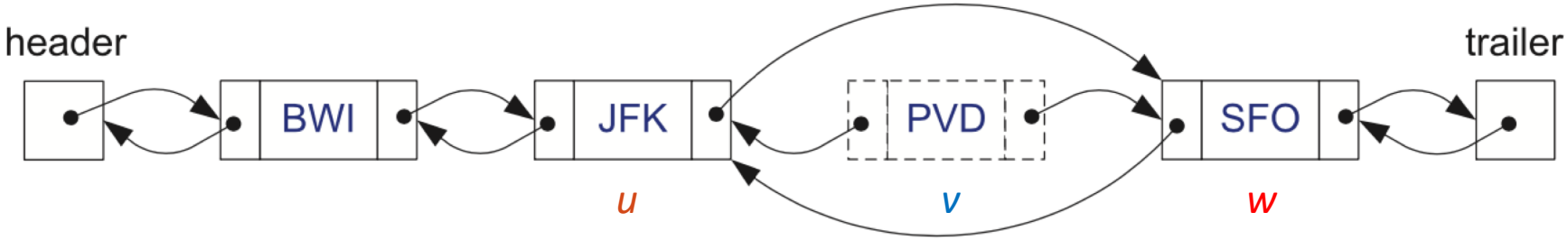


where *v* is the node JFK, *w* is the node PVD, and *z* is the new node BWI. Observe that this process works if *v* is any node ranging from the header to the node just prior to the trailer!

DOUBLY LINKED LIST: REMOVAL

- Let *u* be the node before *v*, and *w* be the node after *v*. To remove node *v*, we have *u* and *w* point to each other instead of to *v*, *i.e.* ***linking out*** of *v:*
  - Make *w*'s *prev* link point to *u*
  - Make *u*'s *next* link point to *w*
  - Delete node *v*

Example:



DOUBLY LINKED LIST: NODE CLASS DEFINITION

Example:

```
typedef string Elem; // define the type Elem to be string (can be any data type)
class DNode {
private:
        Elem elem;  // node element value of the type defined earlier
        DNode* prev;              // previous node in the list
        DNode* next;               // next node in the list
        friend class DLinkedList;
};
```

DOUBLY LINKED LIST: LINKED LIST CLASS DEFINITION

Example:

```
class DLinkedList {
public:
        DLinkedList();          // constructor
        ~DLinkedList();         // destructor
        bool empty() const;
        const Elem& front() const;
        const Elem& back() const;
        void addFront(const Elem& e);
        void addBack(const Elem& e);
        void removeFront();
        void removeBack();
private:
        DNode* header;
        DNode* trailer;
protected:
        void add(DNode* v, const Elem& e); //insert new node before v
        void remove(DNode* v); };          // remove node v
```

DOUBLY LINKED LISTS: LINKED LIST CLASS - METHODS

Example (some of the methods, rest in textbook):

```
DLinkedList::DLinkedList() {          // constructor
header = new DNode;
trailer = new DNode;
header->next = trailer;
trailer->prev = header; }

DLinkedList::~DLinkedList() {         // destructor
while (!empty()) removeFront();
delete header;
delete trailer;  }

const Elem& DLinkedList::front() const {   // get the front element
return header->next->elem; }

void DLinkedList::addFront(const Elem& e) {  // add to front of list
add(header->next, e); }
```

DOUBLY LINKED LISTS: LINKED LIST CLASS – ADD NODE

Example:

```
// insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
// create a new node for e
DNode* u = new DNode;
u->elem = e;
// link u in between v and v->prev
u->next = v;
u->prev = v->prev;
v->prev->next = v->prev = u; // make u the next for the
                             // node before v
}
```

DOUBLY LINKED LISTS: LINKED LIST CLASS – REMOVE NODE

Example:

```
// remove node v
void DLinkedList::remove(DNode* v) {
DNode* u = v->prev; // the previous node
DNode* w = v->next; // the next node

// unlink v from list
u->next = w;
w->prev = u;
delete v;

}
```

DOUBLY LINKED LIST: REVERSING A LINKED LIST

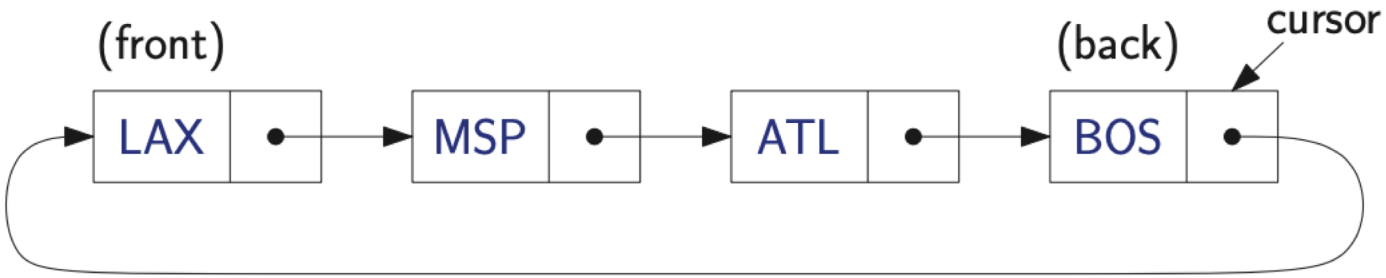
- The proposed approach involves the following:
  1. Copying the contents of the list in reverse order into a temporary list
  2. Copying the contents of the temporary list back into the original list

DOUBLY LINKED LIST: REVERSING A LINKED LIST

```
void listReverse(DLinkedList& L) {
 DLinkedList T;              // temporary list

 // reverse L into T
 while (!L.empty()) {
       string s = L.front();
       T.addFront(s);
       L.removeFront();

 }

 // copy T back to L
 while (!T.empty()) {
       string s = T.front();
       T.removeFront();
       L.addBack(s);

}
```

CIRCULARLY LINKED LISTS

- Has nodes similar to singly linked list (each has one pointer)
- The nodes are linked into a cycle (has no beginning or end)
  - **Cursor** node allows us to have a place to start from



  - ***Back:*** is the element referenced by the cursor
  - ***Front:*** is the element immediately following the circular order

CIRCULARLY LINKED LIST: NODE CLASS DEFINITION

Example:

```
typedef string Elem; // define the type Elem to be string (can be any data type)
class CNode {
private:
        Elem elem;  // node element value of the type defined earlier
        CNode* next;                  // next node in the list
        friend class CircleList;
};
```

CIRCULARLY LINKED LIST: LINKED LIST CLASS DEFINITION

Example:

```
class CircleList {
public:
      CircleList();   // constructor
     ~CircleList();   // destructor
     bool empty() const;
     const Elem& front() const; // get element at cursor
     const Elem& back() const; // get element following cursor
     void advance();                // advance cursor
     void add(const Elem& e);    // add after cursor
     void remove();                 // remove node after cursor
private:
     CNode* Cursor;  // the cursor
```

CIRCULARLY LINKED LISTS: LINKED LIST CLASS – METHODS

Example (for some of the methods, rest in textbook):

```
bool CircleList::empty() const                  // is list empty?
{ return cursor == NULL; }

const Elem& CircleList::back() const           // get element at cursor
{ return cursor->elem; }

const Elem& CircleList::front() const          // get element following cursor
{ return cursor->next->elem; }

void CircleList::advance()                  // advance cursor
{ cursor = cursor->next; }
```

CIRCULARLY LINKED LISTS: LINKED LIST CLASS – ADD NODE

Example:

```
void CircleList::add(const Elem& e) { // add after cursor
 CNode* v = new CNode;              // create a new node
 v->elem = e;
 if (cursor == NULL) {     // if list is empty
       v->next = v; // v points to itself
       cursor = v;  // cursor points to v
 }
 else {                    // if list is nonempty
       v->next = cursor->next;   // link in v after cursor
       cursor->next = v;
 }
}
```

CIRCULARLY LINKED LISTS: LINKED LIST CLASS – REMOVE NODE

Example:

```
void CircleList::remove() { // remove node after cursor
 CNode* old = cursor->next;           // the node being removed
 v->elem = e;
 if (old == cursor) {   // removing the only node?
      cursor = NULL;    // list is now empty
 }
 else {      // if list is nonempty
      cursor->next = old->next;      // link out the old node
 }
 delete old;  // delete the old node
}
```

RECURSION

RECURSION

- Another way to achieve repetition, other than loops, is through ***recursion***
- Recall Recursion from Intro to CS:
  - It is when a function calls itself

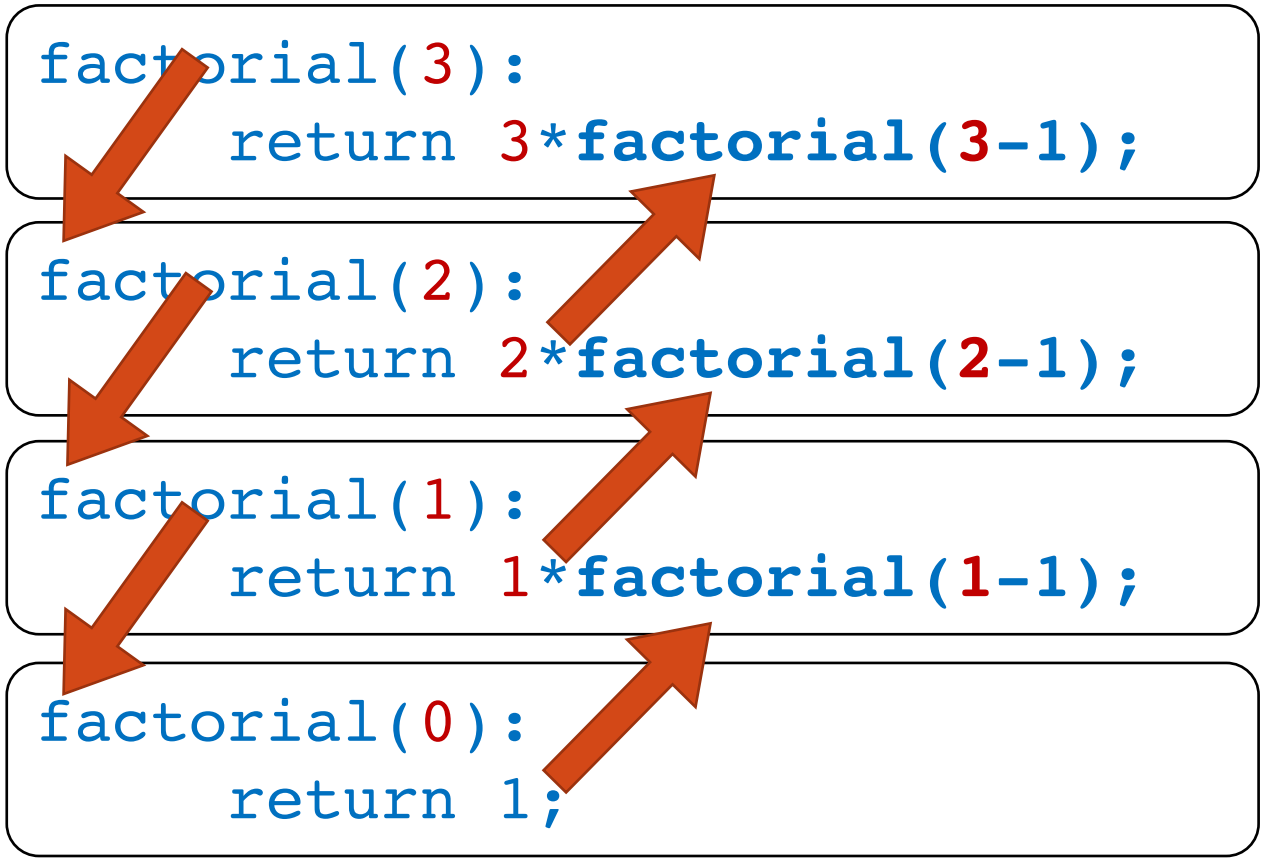
EXAMPLE I: FACTORIAL USING RECURSION

- Factorial of n! is defined as:
  - If n > 0 then n! = 1 x 2 x 3 x 4 x 5 x .... x n ≡ (n-1)!*n
  - If n = 0 then 0! = 1

- **C++ implementation:**

```
int factorial(int n) {
 if (n == 0) {return 1;} // basis case
 // recursive case
 else {return n * factorial(n-1);}
}
```

EXAMPLE I: FACTORIAL - RECURSION TRACE



Returned value:

3*2 = 6

2*1 = 2

1*1 = 1

1

```
int factorial(int n) {
 if (n == 0) {
        return 1;
  }
 else {
        return n*factorial(n-1);
  }
}
```

REVERSE ARRAY VIA RECURSION

TAIL RECURSION

```
void ReverseArray(int A[], int i, int j){
 if (i < j){
      int tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
      ReverseArray(A,i+1,j-1); }
 return; // nothing happens
 // tail recursion: the very last step is a recursive call
}
```

EXAMPLE II: FIBONACCI SERIES

- Mathematically the series can be defined as:
  - If n=0 ⟹ Fib(n)=0
  - If n=1 ⟹ Fib(n)=1
  - If n > 1 ⟹ Fib(n) = Fib(n-1)+Fib(n-2)

- Notice that the Fibonacci definition by default has a recursion

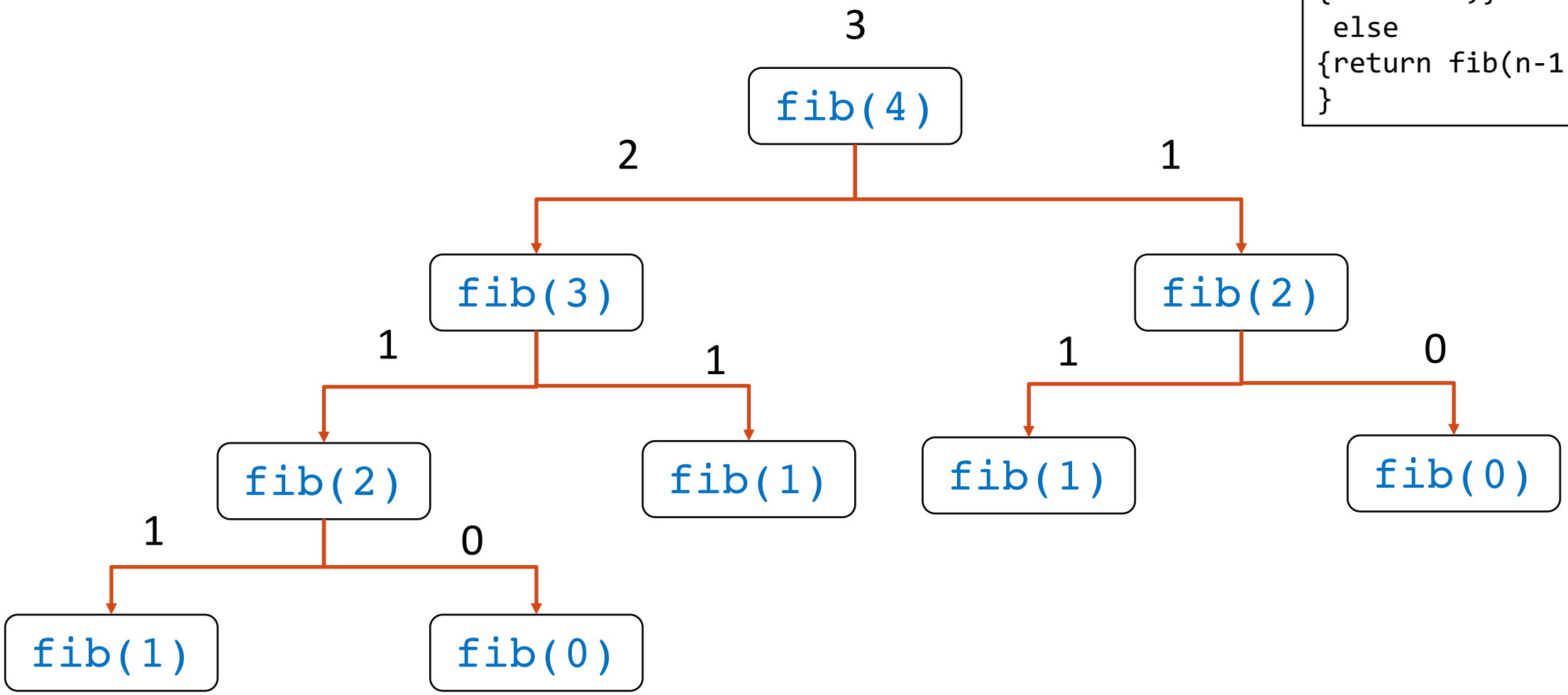
EXAMPLE II: FIBONACCI SERIES

- **C++ implementation:**

```
int fib(int n) {
 if (n == 0 || n == 1) {return n;} // basis case
 // recursive case
 else {return fib(n-1)+fib(n-2);}
}
```

EXAMPLE II: FIBONACCI SERIES

```
int fib(int n) {
 if (n == 0 || n == 1)
{return n;}
 else
{return fib(n-1)+fib(n-2);}
}
```



TYPES OF RECURSION

1. **Linear recursion**
   - At most one recursive call is made each time it is called
   - E.g. Factorial
2. **Binary recursion**
   - Two recursive calls are made each time it is called
   - E.g. Fibonacci Series
3. **Multiple recursion**
   - Multiple recursive calls are made each time it is called

NEXT LECTURE

- Continue with recursion
- Analysis Tools

- Readings:
  - Chapter 4