CSIS/COMP 1117B
Computer Programming

Linked Lists

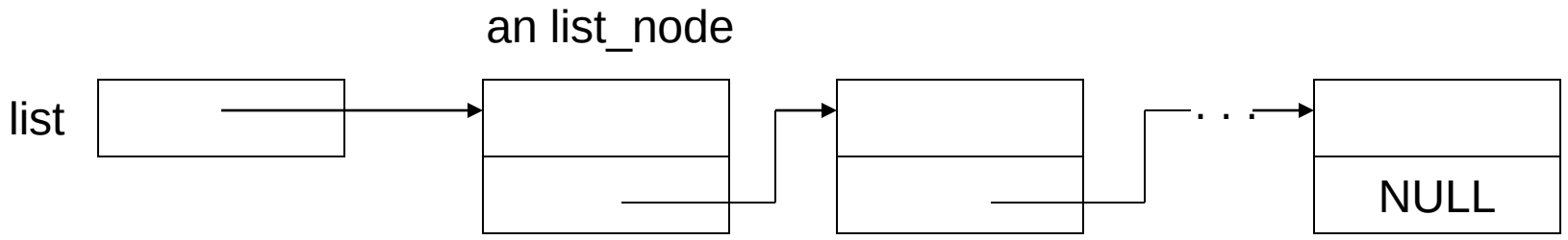
Linked Lists

- A linked list is a ***data structure*** where data objects are accessed ***sequentially***.
- A linked list is a chain of nodes; each **node** has two parts: a data part and a link part.
- To access an individual node, we need to start from the **head** of the list and *chase down* the links until we reach the node of interest.
- A linked list is a dynamic data structure because it can grow and shrink and its size is only limited by the amount of available storage.

Node Declaration

- The following declares a type for nodes of a linked list of int:

```
struct list_node {
    int  data;
    list_node *next;
} *list;
```



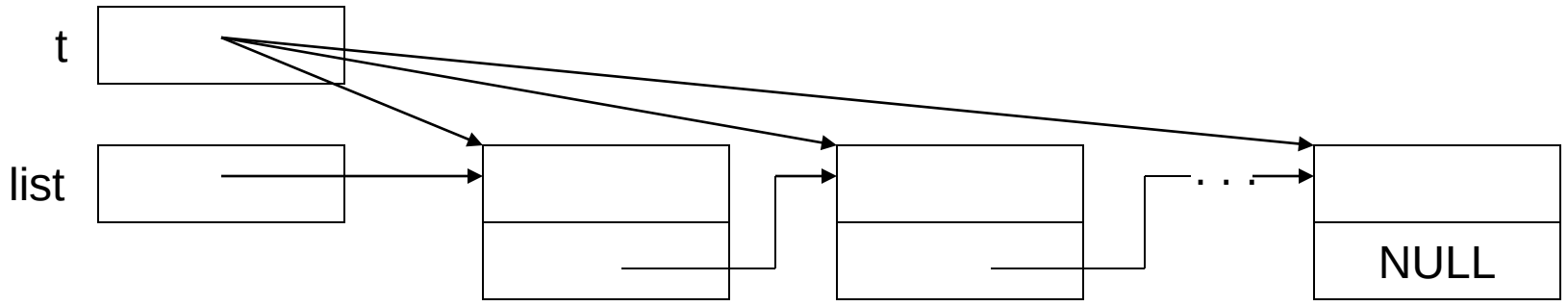
Operations of Linked Lists

- Traversing a list
- Searching for a node
- Insertion at the head
- Insertion after a node
- Deletion at the head
- Deletion after a node

Traversing a list

- **Traversing** a list means to go through the entire list from beginning to end.

```
for (list_node *t = list; t != NULL; t = t->next) {
    . . . // perform operation on contents of (*t)
}
```



Searching for a Node

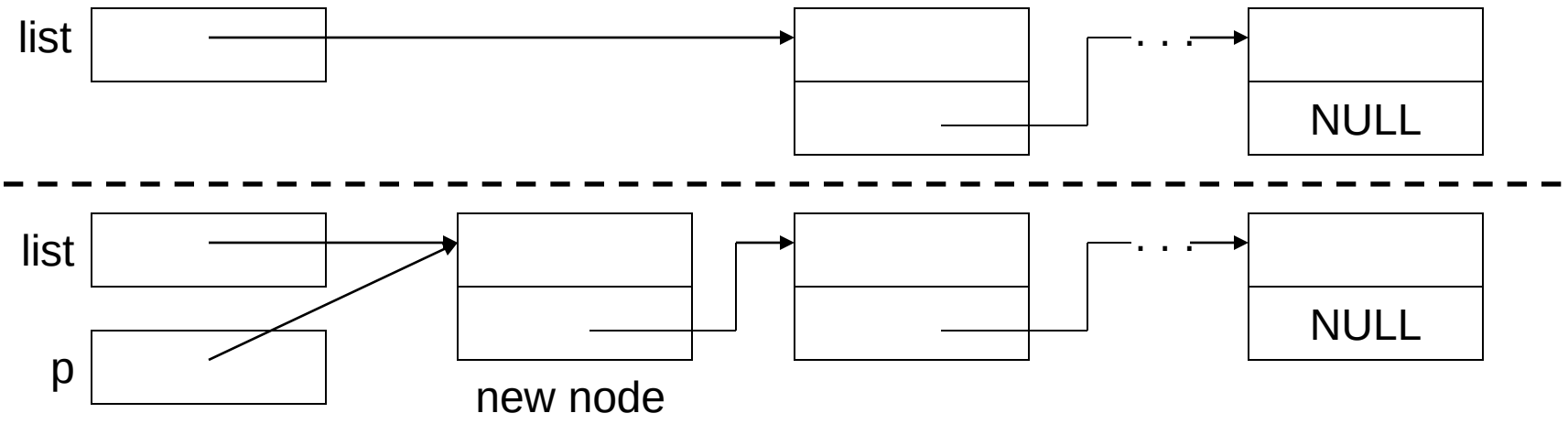
- **Searching** through the list from beginning to end until a node whose contents match a given key is found or that the end has been reached but none of the nodes matches the given key.
- Note that this is a special case of traversing a list.

```
list_node *t = list;
for ( ; t != NULL; t = t->next) {
    if (t->data == key) {
        // t points at the matching node
        break;
    }
}
// the search fails if t is NULL here
```

Insertion at the Head

- **Inserting** a node as the **new head** of a list.

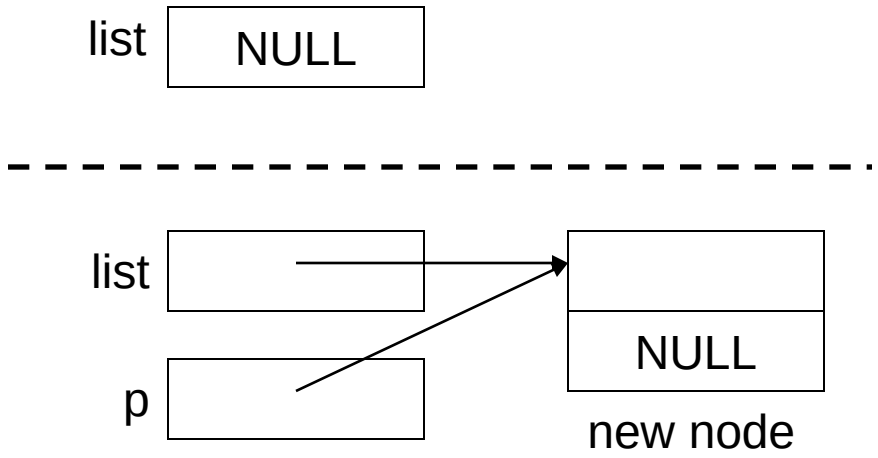
```
list_node *p = new list_node;
p->next = list;
list = p;
```



What If the List is Empty?

- The same code still works in this case.

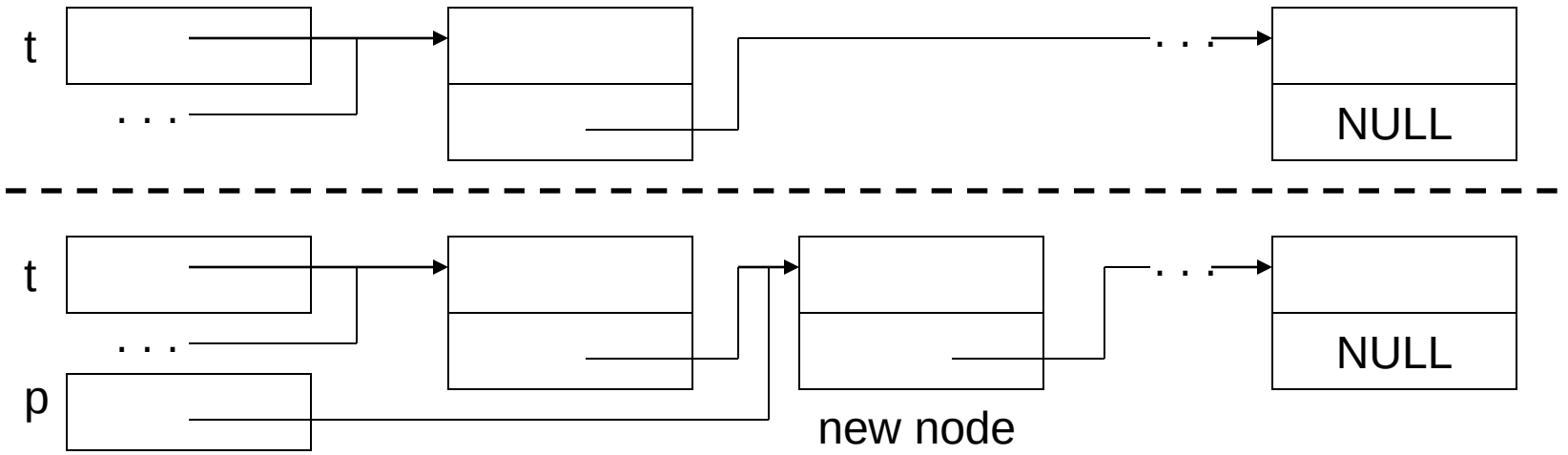
```
list_node *p = new list_node;
p->next = list;
list = p;
```



Insertion after a Node

- Inserting a new node ***after*** the one pointed at by a pointer t.

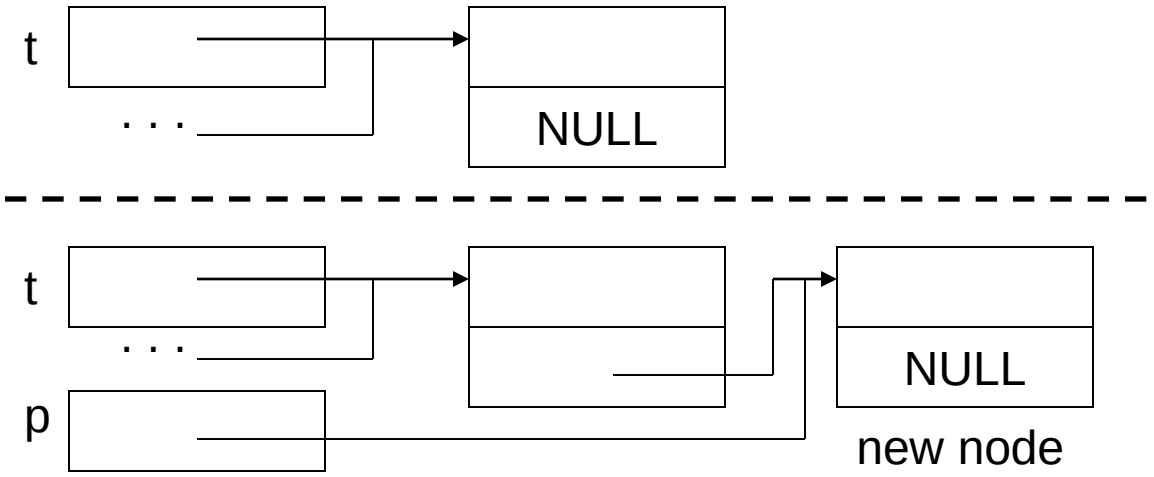
```
list_node *p = new list_node;
p->next = t->next;
t->next = p;
```



What If It is the Last Node?

- The same code still works in this case.

```
list_node *p = new list_node;
p->next = t->next;
t->next = p;
```

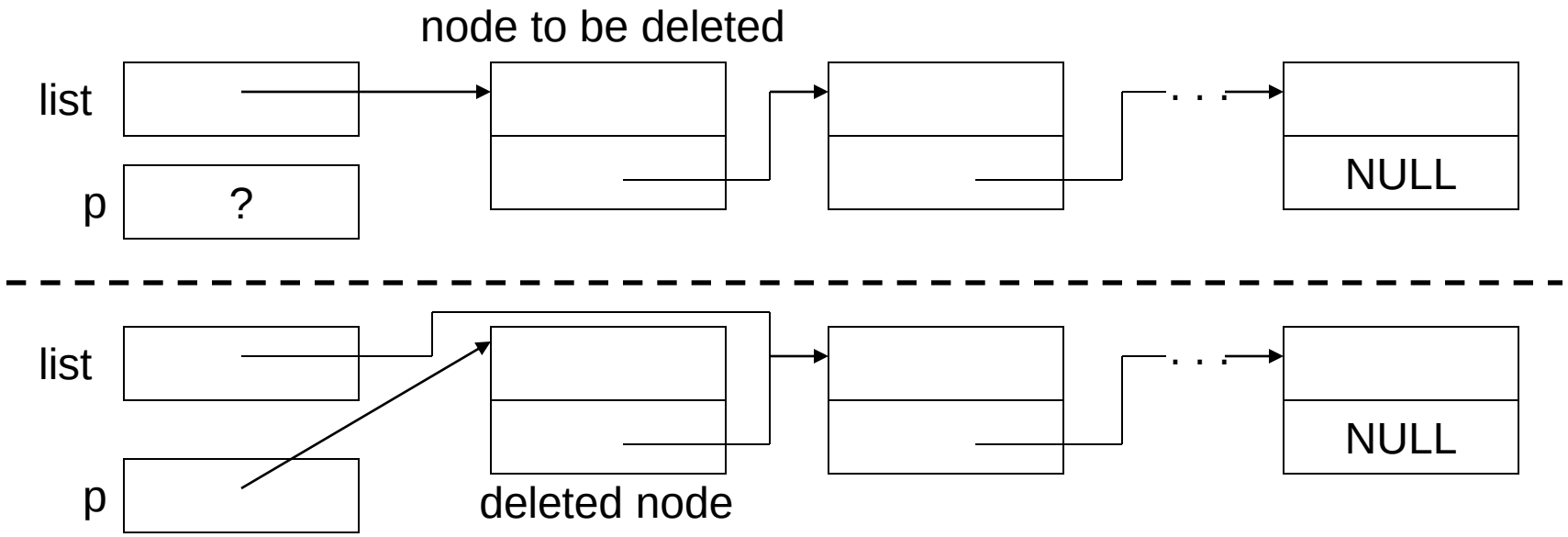


Deletion at the Head

- **Deleting** the node pointed at by list, i.e., the first node.

  list_node *p = list;

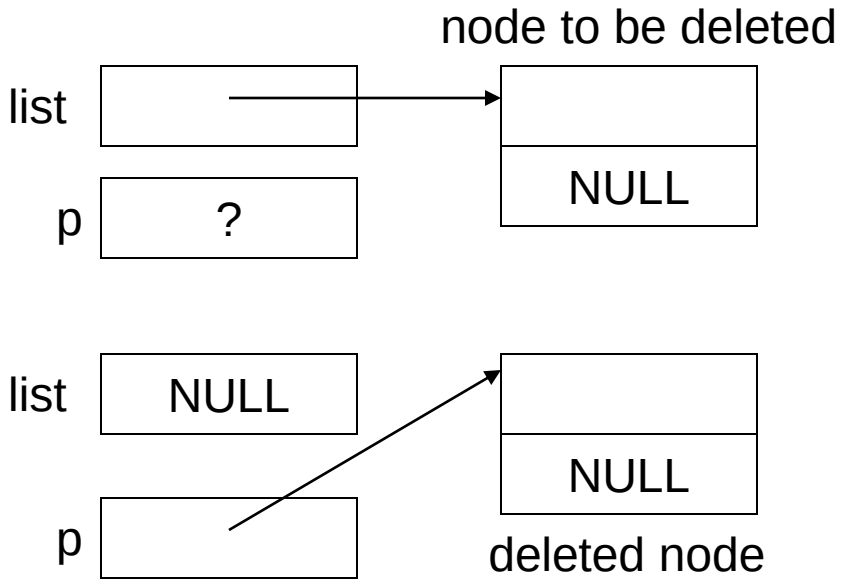
  list = list->next;



What If It is the Only Node?

- The same code still works if the node to be deleted is the only node in the list.

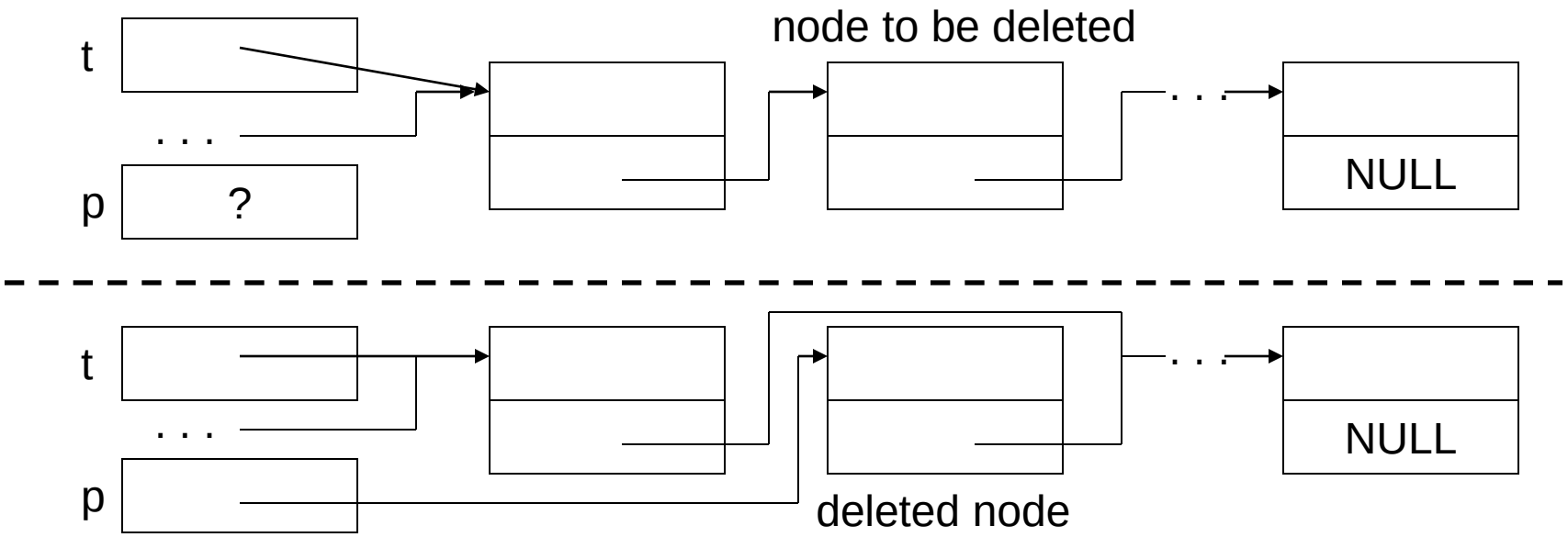
```
list_node *p = list;
list = list->next;
```



Deletion after a Node

- Deleting the node ***following*** the node pointed at by a pointer t.

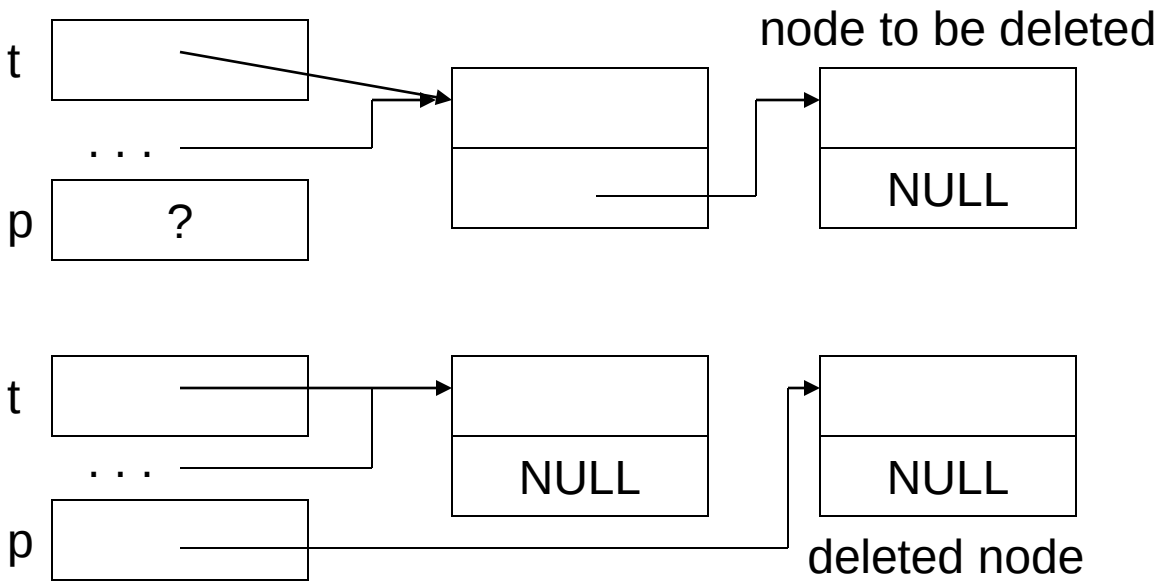
```
list_node *p = t->next;
t->next = p->next;
```



What If It is the Last Node?

- The same code still works if the node to be deleted is the last node in the list.

```
list_node *p = t->next;
t->next = p->next;
```



Insertion in Order

- Assume that the nodes are ***linked in order*** of their contents, say in ascending order; that is, the contents of the first node is ***smaller*** than that of the second and so on.

```
void insert_in_order(list_node *&list, int key) {
    list_node *t, *p = new list_node;
    p->data = key;
    if (list == NULL || list->data >= key) {
        // insert node at head of list
        p->next = list;
        list = p;
        return;
    }
    for (t = list; t->next != NULL; t = t->next)
        if (t->next->data >= key) break;
    //  insert after node pointed at by t
    p->next = t->next;
    t->next = p;
}
```

Recursive Version

```
void insert_in_order(list_node *&list, int key) {
    if (list == NULL || list->data >= key) {
        // insert node at head of list
        list_node *p = new list_node;
        p->data = key;
        p->next = list;
        list = p;
    } else insert_in_order(list->next, key);
}
```

Deletion of Node with Matching Key

- The following is modified from the code for searching:

```
// delete_key return pointer to the deleted node, NULL if not found
list_node *delete_key(list_node *&list, int key) {
    if (list == NULL) return NULL;
    if (list->data == key) {          // delete first node of list
        list_node *t = list;
        list = list->next;
        return t;
    }
    for (list_node *t = list; t->next != NULL; t = t->next) {
        if (t->next->data == key) {  // delete node following the one pointed at by t
            list_node *p = t->next;
            t->next = p->next;
            return p;
        }
    }
    return NULL;
}
```

Recursive Version

```
// delete_key return pointer to the deleted node,
//    NULL if not found
list_node *delete_key(list_node *&list, int key) {
    if (list == NULL) return NULL;
    if (list->data == key) {           // delete first node of list
        list_node *t = list;
        list = list->next;
        return t;
    } else return delete_key(list->next, key);
}
```

A Smaller Example of ADT

- Consider a stack, which is a storage organization with only one access point and two operations: push and pop.
- Can we have a stack of activation records?
  - push will add one activation record to the stack
  - pop will remove one activation record from the stack
  - *is it possible to define a data type for activation records?*
  - **need the ability to treat entities that are of different types to be compatible for operations that do not depend on the details of the type -- polymorphism**

Template Class

- You have already seen a template class before; vector is a template class parameterized on the type of objects making up the vector.
- A template class takes one parameter (which is a class) and the class parameter can be used in the definition of the template class just like any class name.
- Template classes support ***parametric polymorphism***.
- The form of a template class definition:

  template ‘<‘ class <type> ‘>’ <class definition> ‘;’

A Stack Template Class

```
template <class any> class stack {
private:
    struct node {
        any*    contents;
        node*   next;
      };
    node*       top;
public:
    void    push(any *s) { . . . }
    any*   pop() { . . . }
    bool   empty() { . . . }
    stack() { top = NULL; }
    ~stack() {
        node *t;
        while (top != NULL) {
            t = top;
            top = top->next;
            delete t->contents;
            delete t;
        }
};
```

The Member Functions

```
void push(any* s) {
    node*  t = new node;
    t->next = top;
    t->contents = s;
    top = t;
}
any*  pop() {
    if (top == NULL) return NULL;
    node* t = top;
    any*   s = top->contents;
    top = top->next;
    delete t;
    return s;
}
bool empty() { top == NULL; }
```

A Stack of string

```
int main() {
    stack<string> *str_stack = new stack<string>;
    string  *s = new string(“first string”);
    str_stack->push(s);
    s = new string(“second string”);
    str_stack->push(s);
    s = new string(“third string”);
    str_stack->push(s);
    s = new string(“one final string”);
    str_stack->push(s);
    cout << “start clearing stack\n”;
    while ( ! str_stack->empty() ) {
        s = str_stack->pop();
        cout << *s << endl;
        delete s;
    }
}
```

What about Queues?

- A queue is a storage whose contents are accessed in a first-in-first-out (FIFO) manner
  - contrast with a stack which is last-in-first-out
- A queue can be implemented as a linked list with an additional pointer pointing to the last node
  - insertion (***enqueue***) is done at the ***tail*** of the list
    - if tail is NULL, insertion into an empty list; otherwise insertion after node pointed at by tail
  - deletion (***dequeue***) is done at the ***head*** of the list
    - delete node pointed at by head if head is not NULL

A Queue Template Class

```
template <class any> class queue {
private:
    struct node {
        any*    contents;
        node*  next;
      };
    node    *head, *tail;
public:
    void   enqueue(any *s) { . . . }
    any*  dequeue() { . . . }
    bool  empty() { . . . }
    queue() { head = NULL;  tail = NULL; }
    ~queue() {
        node *t;
        while (head != NULL) {
            t = head;
            head = head->next;
            delete t->contents;
            delete t;
        }
};
```

The Member Functions

```
void enqueue(any* s) {
    node*  t = new node;
    t->next = NULL;
    t->contents = s;
    if (tail == NULL) head = t;  // first node added
    else tail->next = t;
    tail = t;
}
any*  dequeue() {
    if (head == NULL) return NULL;
    node* t = head;
    any*   s = head->contents;
    head = head->next;
    if (head == NULL) tail = NULL;
    delete t;
    return s;
}
bool empty() { head == NULL; }
```