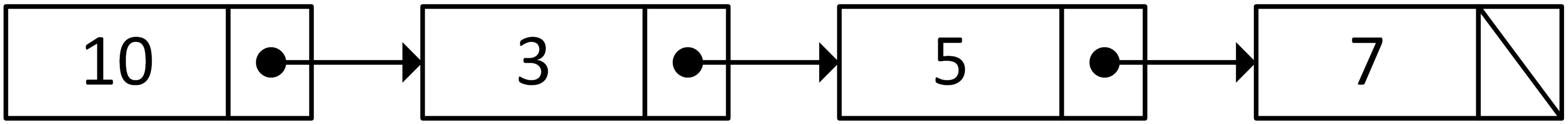
Linked Data Structures

Readings: CP:AMA 17.5

Linked lists

Racket's list type is more commonly known as a ***linked list***.



Each ***node*** contains an ***item*** **(first)** and a *link* to the ***next*** node **(rest)**.

There is no "official" way of implementing a linked list in C.

In this unit we present a typical linked list structure and three different approaches for working with this structure.

In general, a linked list is a pointer to a *linked list node* (`llnode`).

```
struct llnode {
  int item;
  struct llnode *next;
};
```

A C structure can contain a *pointer* to its own structure type. This is the first **recursive data structure** we have seen in C.

If a linked list is a pointer to a node, then an empty list is represented by a `NULL` pointer.

In the following slides we introduce three different *"approaches"* for working with linked lists:

- a **functional** (Racket-like) approach,
- an **imperative** (C-like) approach, and
- a **wrapper** approach (an extension of imperative).

These are not well-established distinctions, but will help illustrate different **paradigms** for working with linked lists.

These approaches are not mutually exclusive. They are compatible (they all use `llnode`s) and can be combined.

Linked lists: functional approach

Helper functions to create a *“functional”* (Racket-like) atmosphere:

```
int first(struct llnode *lst) {
  assert (lst != NULL);
  return lst->item;
}

struct llnode *rest(struct llnode *lst) {
  assert (lst != NULL);
  return lst->next;
}

struct llnode *empty(void) {
  return NULL;
}

bool is_empty(struct llnode *lst) {
  return lst == empty();
}
```

At the heart of the functional approach is the `cons` function.

In Racket, `cons` acquires dynamic memory (similar to C's `malloc`).

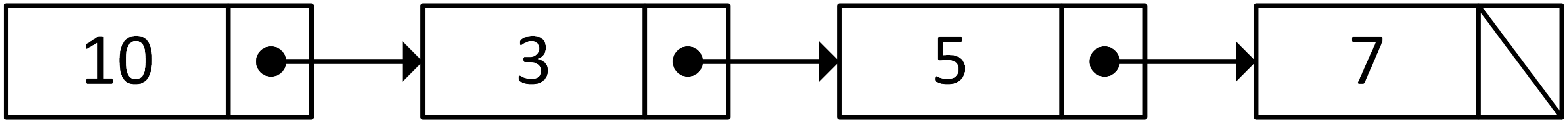
Our C `cons` `return`s a **new** node that *links* to the rest.

```
struct llnode *cons(int f, struct llnode *r) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = f;
  new->next = r;
  return new;
}
```

This is very similar to how Racket's `cons` is implemented.

We can use our new C `cons` function the same way we used our Racket `cons` function.

```
struct llnode *my_list = cons(10, cons(3, cons(5,
                              cons(7, empty()))));
```



We can also use `cons` in different ways (*e.g.*: with mutation).

```
struct llnode *my_list  = empty();
my_list = cons(7, my_list);
my_list = cons(5, my_list);
my_list = cons(3, my_list);
my_list = cons(10, my_list);
```

Using the functional approach, we can write recursive functions that closely mirror their Racket equivalents.

```
int length(struct llnode *lst) {
  if (is_empty(lst)) {
    return 0;
  } else {
    return 1 + length(rest(lst));
  }
}
```

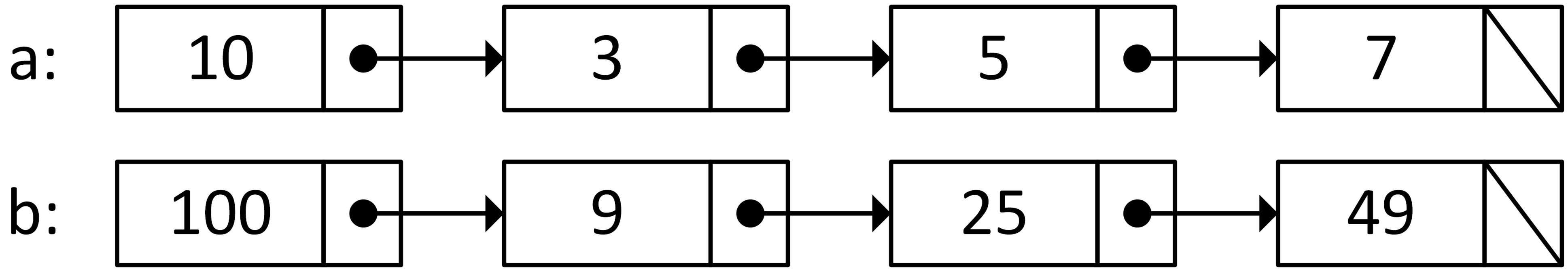
It is also possible to write some functions iteratively.

```
int length_iterative(struct llnode *lst) {
  int length = 0;
  while (!is_empty(lst)) {
    length++;
    lst = rest(lst);
  }
  return length;
}
```

It may not occur to some Racket programmers that most functions that *produce* a list `cons`truct a **new list**.

```
(define (sqr-list lst)
  (cond [(empty? lst) empty]
        [else (cons (* (first lst) (first lst))
                    (sqr-list (rest lst)))]))
```

```
(define a '(10 3 5 7))
(define b (sqr-list a))
```



A C function written with a functional approach will also `return` a **new list**.

```
struct llnode *sqr_list(struct llnode *lst) {
  if (is_empty(lst)) {
    return empty();
  } else {
    return cons(first(lst) * first(lst),
                sqr_list(rest(lst)));
  }
}
```

As an exercise, try writing a non-recursive (iterative) version of `sqr_list` that returns a new list (it’s tricky).

Since we have been working with imperative C, it might now seem more “natural” that a `sqr_list` function would square (or *mutate*) each item in the list.

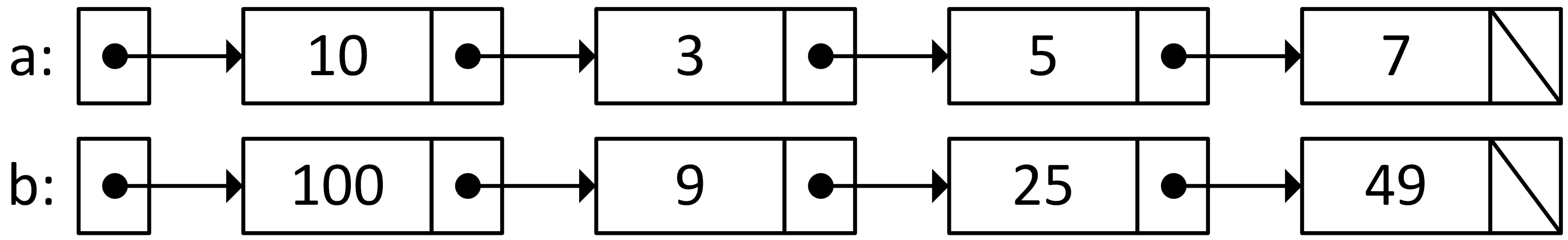
However, this is not a functional (Racket-like) approach. We will introduce an imperative version of `sqr_list` shortly.

In Racket, lists are immutable, and there is a special `mcons` function to generate a mutable list.

In the Scheme language, lists are mutable. This is one of the significant differences between Racket and Scheme.

To correctly use the `sqr_list` function, the result should be stored in a separate variable.

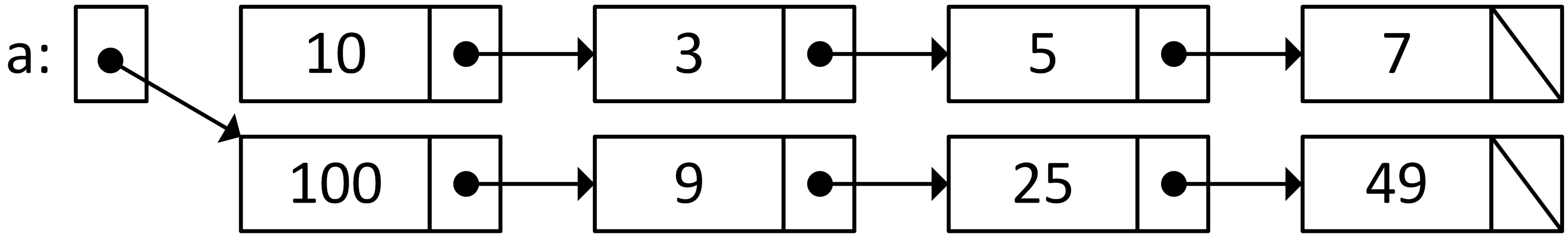
```
struct llnode *a = cons(10, cons(3, cons(5, cons(7, empty()))));
struct llnode *b = sqr_list(a);
```



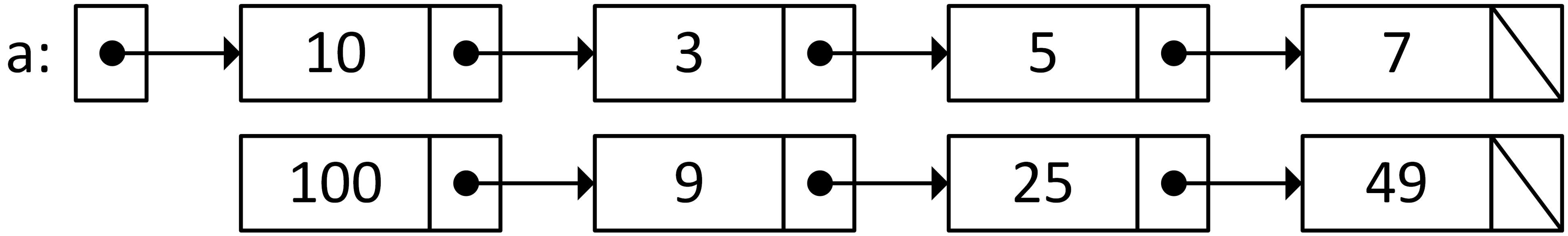
Unfortunately, if the function is misunderstood or used **incorrectly**, it can create a **memory leak**.

The following two statements each create a memory leak:

```
a = sqr_list(a);  // original list is lost
```



```
sqr_list(a);      // new list is lost
```



Even the `cons` function can cause a similar problem:

```
struct llnode *a = cons(7, empty());

cons(5, a);  // memory leak
```

In Racket, this was not a concern because Racket automatically handles memory leaks.

(see the note on **garbage collection** in the previous module)

In Racket, we are also not concerned about `free`ing a list.

However, this is necessary in C.

```
void free_list(struct llnode *lst) {
  if (lst != NULL) {
    free_list(rest(lst));
    free(lst);
  }
}
```

Note that it is important to `free` the rest of the list before `free`ing the first node. After `free(lst)`, any use of `lst` is invalid.

Both of these equivalent, *iterative* `free_list` functions require a "backup" pointer to avoid accessing invalid memory after `free`.

```
void free_list_iterative_1(struct llnode *lst) {
  while (lst != NULL) {
    struct llnode *backup = lst;
    lst = rest(lst);
    free(backup);
  }
}

void free_list_iterative_2(struct llnode *lst) {
  while (lst != NULL) {
    struct llnode *backup = rest(lst);
    free(lst);
    lst = backup;
  }
}
```

This "backup" technique (using temporary pointers) will be used more frequently when we introduce the *imperative* approach.

To further illustrate how `cons` is used, consider the `insert` function we used in *insertion sort* (insert into a *sorted* list of numbers).

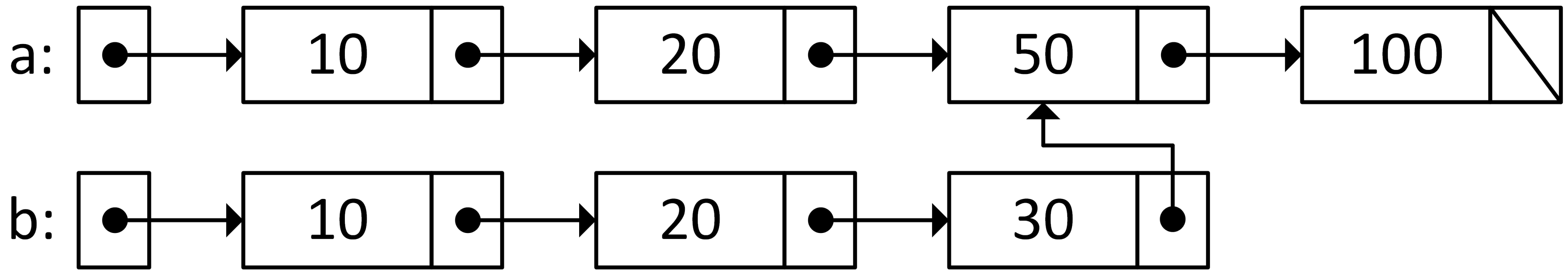
```
(define (insert n slst)
  (cond
    [(empty? slst) (cons n empty)]
    [(<= n (first slst)) (cons n slst)]
    [ else (cons (first slst) (insert n (rest slst)))]))
```

```
struct llnode *insert(int n, struct llnode *slst) {
  if (is_empty(slst)) {
    return cons(n, empty());
  } else if (n <= first(slst)) {
    return cons(n, slst);
  } else {
    return cons(first(slst), insert(n, rest(slst)));
  }
}
```

Consider this example:

```
struct llnode *a = cons(10,cons(20,cons(50,cons(100,empty()))));
struct llnode *b = insert(30, a);
```

The lists will **share the last two nodes**.



Freeing either list will cause a crash (invalid memory access) when the other is accessed.

```
free_list(a);
free_list(b);   // INVALID
```

This problem can occur with just the `cons` function.

```
struct llnode *a = cons(7, empty());
struct llnode *b = cons(5, a);
a = cons(3, a);

free_list(a);
free_list(b);  // INVALID
```

Once again, Racket automatically handles `free`ing memory, so this is not a concern in Racket.

Using a functional (Racket-like) approach is appropriate for some applications, but it is not well suited for mutating lists or when some specific memory management issues arise.

Linked lists: imperative approach

In an "**imperative**" (C-like) approach to linked lists, individual nodes are manipulated and changed (mutated) directly.

An imperative list function may *change* a list instead of generating a new list.

For example, the following `sqr_list` function *changes* the items in the list.

```
void sqr_list(struct llnode *lst) {
  while (lst != NULL) {
    lst->item *= lst->item;
    lst = lst->next;
  }
}
```

If a new list is desired, a *copy* the original list can be made.

```
struct llnode *copy_list(struct llnode *lst) {
  if (lst == NULL) return NULL;
  return cons(lst->item, copy_list(lst->next));
}
```

The copy can then be changed.

```
struct llnode *orig = cons(10, cons(3, cons(5,
                                cons(7, empty()))));

struct llnode *dup = copy_list(orig);
sqr_list(dup);
```

As an alternative to the `cons` function, we will write a function that inserts a new node at the start (front) of an existing list.

For example, we would like to have the following code sequence:

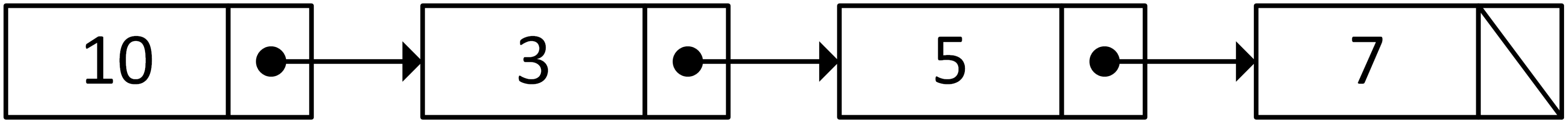
```
struct llnode *lst = NULL;

add_to_front(7, lst);   // won't work
add_to_front(5, lst);   // won't work
```

However, to have a function change a variable (*i.e.:* `lst`), we need to pass a *pointer* to the variable.

```
add_to_front(7, &lst);
add_to_front(5, &lst);
```

Since `lst` is already a pointer, we will need to pass a *pointer to a pointer*.

```
void add_to_front(int n, struct llnode **ptr_front) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = n;
  new->next = *ptr_front;
  *ptr_front = new;
}
```



The following code generates the above list.

```
struct llnode *lst = NULL;
add_to_front( 7, &lst);
add_to_front( 5, &lst);
add_to_front( 3, &lst);
add_to_front(10, &lst);
```

With the imperative approach, nodes can also be *removed*.

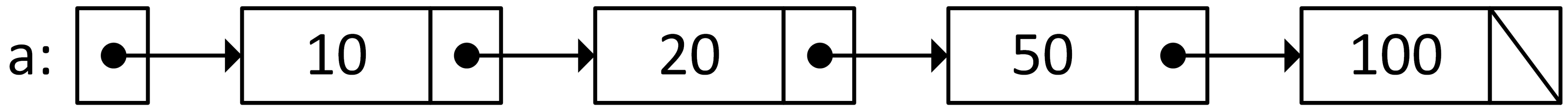
```
int remove_from_front(struct llnode **ptr_front) {
  struct llnode *front = *ptr_front;
  int retval = front->item;
  *ptr_front = front->next;
  free(front);
  return retval;
}
```

Instead of `return`ing nothing (`void`), it is more useful to `return` the value of the item being removed.

This function `inserts` a node into an *existing* sorted list.

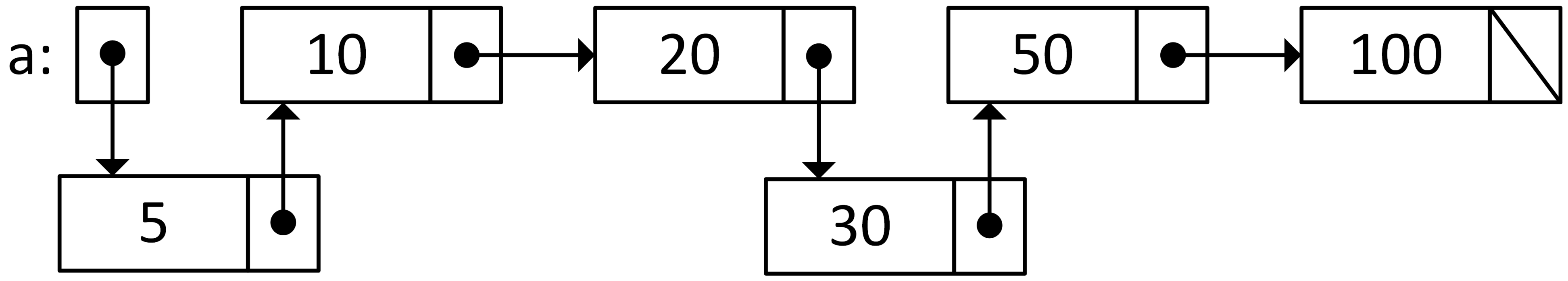
```
void insert(int n, struct llnode **ptr_front) {
    struct llnode *cur = *ptr_front;
    struct llnode *prev = NULL;
    while ((cur != NULL) && (n > cur->item)) {
      prev = cur;
      cur = cur->next;
    }
    struct llnode *new = malloc(sizeof(struct llnode));
    new->item = n;
    new->next = cur;
    if (prev == NULL) { // new first node
      *ptr_front = new;
    } else {
      prev->next = new;
    }
  }
}
```

If we revisit the previous example:



The list after the following two inserts will be:

```
insert( 5, &a);
insert(30, &a);
```



The imperative approach is still susceptible to misuse.

```
struct llnode *a = NULL;
add_to_front(5, &a);
struct llnode *b = a;
add_to_front(7, &a);
add_to_front(9, &b);
```

In the above example, freeing either list a or b will make the other list invalid.

Linked lists: wrapper approach

In this final approach, we "**wrap**" each list so it is inside of another structure. The definition of a linked list changes slightly to be a pointer to this new structure, and not just a single node.

```
struct llist {
  struct llnode *front;
};
```

This may seem insignificant, but it has 3 advantages:

- it has a cleaner interface (avoiding any pointer-to-pointer awkwardness),
- it is less susceptible to misuse, and
- the wrapper structure can store additional information.

A wrapper approach would typically have a “create” and a “destroy” function.

```
struct llist *create_list(void) {
  struct llist *lst = malloc(sizeof(struct llist));
  lst->front = NULL;
  return lst;
}

void destroy_list(struct llist *lst) {
  free_list(lst->front);
  free(lst);
}
```

Many of the previous functions introduced can be “wrapped” inside of another function.

```
void add_to_front(int n, struct llist *lst) {
  previous_add_to_front(n, &lst->front);
}
```

Overall, this provides a cleaner interface.

```
struct llist *lst = create_list();
add_to_front(5, lst);
add_to_front(7, lst);
destroy_list(lst);
```

The significant advantage of the wrapper approach is that **additional information** can be stored in the list structure.

The run-times of the previous `length` functions are $O(n)$.

However, we can store (or "cache") the length of *in the list structure*, so the length can be retrieved in $O(1)$ time.

```
struct llist {
  struct llnode *front;
  int length;
};

// TIME: O(1)
int length(struct llst *lst) {
  return lst->length;
}
```

Naturally, other functions would have to update the `length` when necessary. For example:

```
struct llist *create_list(void) {
  struct llist *lst = malloc(sizeof(struct llist));
  lst->front = NULL;
  lst->length = 0;  // *****NEW
  return lst;
}

void add_to_front(int n, struct llist *lst) {
  previous_add_to_front(n, &lst->front);
  lst->length++;    // *****NEW
}
```

Another common addition to a linked list structure is a `back` pointer that points to the *last node* in the list.

This allows for an $O(1)$ `add_to_back` function.

```
void add_to_back(int n, struct llist *lst) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = n;
  new->next = NULL;
  if (lst->length == 0) { // empty list
    lst->front = new;
    lst->back = new;
  } else {
    lst->back->next = new;
    lst->back = new;
  }
  lst->length++;
}
```

Again, other functions would also have to support `back`.

By working with a *list* structure instead of directly working with *node* structures, there is less chance for misuse.

However, the wrapper approach introduces entirely new ways that the information can be corrupted.

For example, what if the `length` field does not accurately reflect the true length?

Whenever the same information is represented in data in more than one way, it is susceptible to *integrity* (consistency) issues.

For example, a naïve user may think that the following statement will remove all of the nodes from the list:

```
lst->length = 0;
```

A fourth *“destructive”* approach uses a Racket-like programming interface (functions produce *new* lists), but each list passed to a function may be destroyed (`free`d).

```
struct llnode *insert(int n, struct llnode *slst) {
  if (is_empty(slst)) {
    return cons(n, empty());
  } else if (n <= first(slst)) {
    return cons(n, slst);
  } else {
    int f_backup = first(slst);
    struct llnode *r_backup = rest(slst);
    free(slst);
    return cons(f_backup, insert(n, r_backup));
  }
}
```

This approach has been taught in previous offerings of CS136

In summary,

- the **functional** (Racket-like) approach is appropriate when there is no mutation and memory management is not necessary (*i.e.*: when there is a garbage collector),
- the **imperative** (C-like) approach is appropriate when there is mutation or memory management is necessary, and
- the **wrapper** approach is appropriate when additional information is to be stored in the list or when a cleaner interface is desired.

These three *"approaches"* also apply to other linked data structures, including **trees**.

Trees

In previous courses, you have worked with ***trees***, including ***Binary Search Trees (BSTs)***.

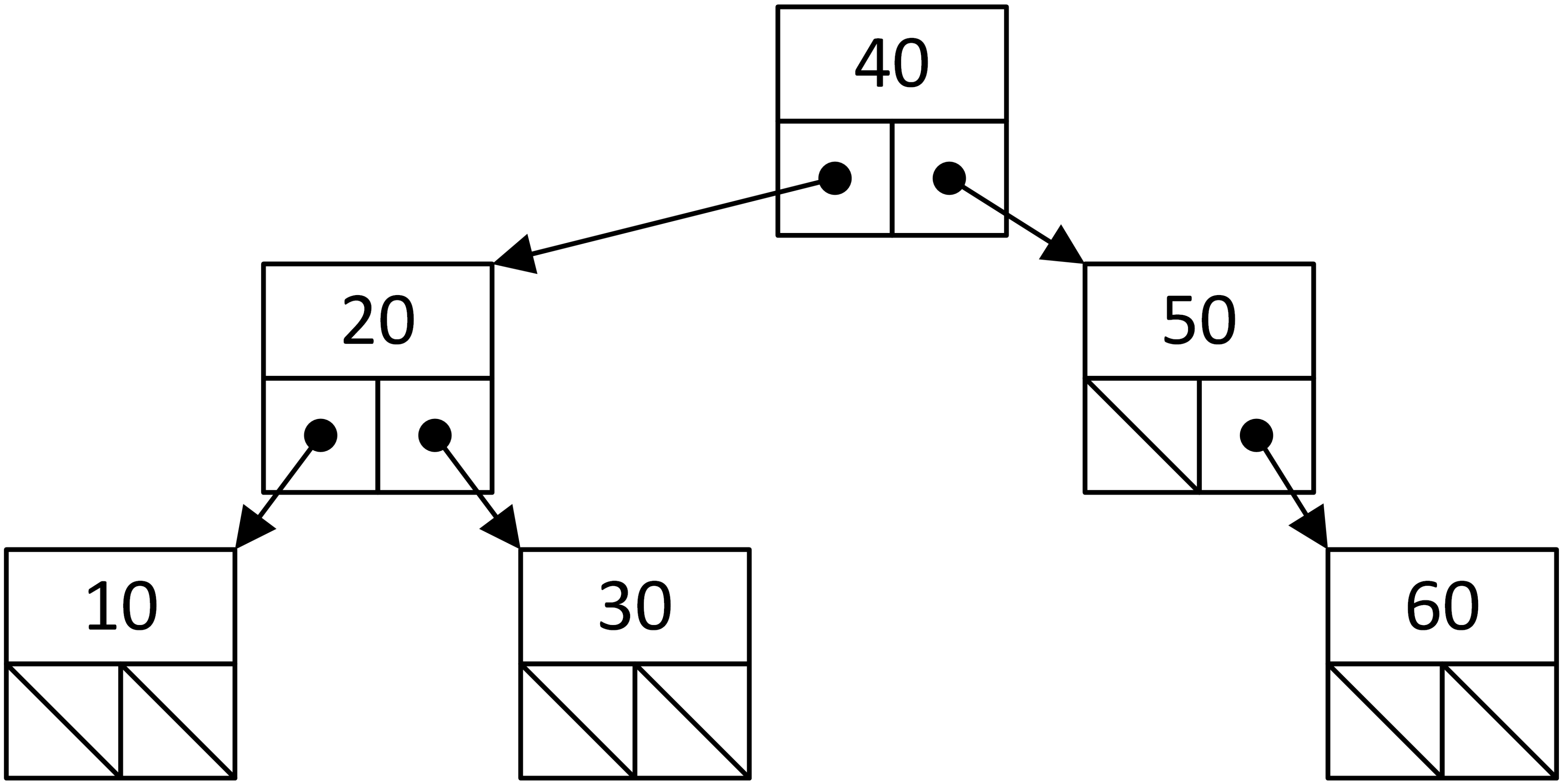
(For a review of tree and BST terminology, see unit 01)

Previously, we used key/value pairs in BSTs. For now, we will only store **single items** in BSTs.

Similar to linked lists, a BST is pointer to *a BST node* (`bstnode`), and an empty tree is `NULL`.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
};
```

The definition of a BST includes the **ordering property**.



Additional tree definitions

The ***depth*** of a node is the number of nodes from the root to the node, including the root. The *depth* of the root node is 1.

The ***height*** of a tree is the maximum *depth* in the tree. The *height* of an empty tree is 0.

For the tree on the previous slide, the *depth* of the node with 50 is 2, and *height* of the tree is 3.

Other courses may use slightly different definitions of *depth* and *height*.

Making new nodes

In Racket, dynamic memory is used to *“make”* a structure.

For example, `(posn 3 4)` (previously `(make-posn 3 4)`) obtains space from the Racket heap for the structure.

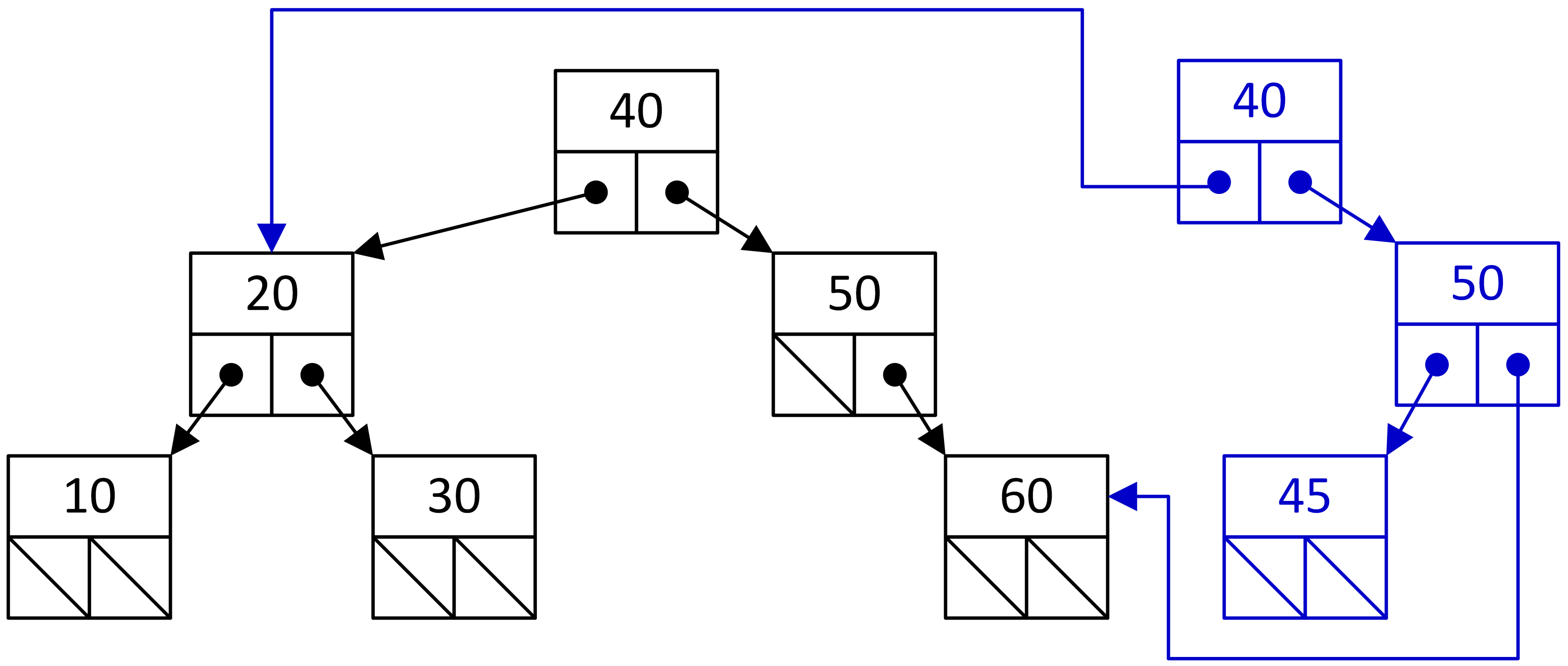
In a **functional** approach, we can similarly *“make”* a bstnode:

```
struct bstnode *make_bstnode(int i, struct bstnode *l,
                                    struct bstnode *r) {

  struct bstnode *new = malloc(sizeof(struct bstnode));
  new->item = i;
  new->left = l;
  new->right = r;
  return new;
}
```

Using a functional approach with BSTs may cause memory management problems similar to the problem shown with the linked list `insert` function.

```
// functional approach
struct bstnode *bst_insert(int i, struct bstnode *t) {
  if (t == NULL) {
    return make_bstnode(i, NULL, NULL);
  } else if (i == t->item) {
    return t;
  } else if (i < t->item) {
    return make_bstnode(t->item,
                        bst_insert(i, t->left), t->right);
  } else {
    return make_bstnode(t->item, t->left,
                        bst_insert(i, t->right);
  }
}
```

```
struct bstnode my_tree = make_bstnode(40,
  make_bstnode(20, make_bstnode(10,NULL,NULL),
                   make_bstnode(30,NULL,NULL)),
  make_bstnode(50,NULL, make_bstnode(60,NULL,NULL)));

struct bstnode new_tree = bst_insert(45, my_tree);
```



An imperative approach can be used to *change* an *existing* tree.

Like with linked lists, we must pass a *pointer to a pointer* to the root BST node.

```
// imperative approach
void bst_insert(int i, struct bstnode **ptr_root) {
  bstnode *t = *ptr_root;
  if (t == NULL) { // tree is empty
    *ptr_root = make_bstnode(i, NULL, NULL);
  } else if (t->item == i) {
    return;
  } else if (i < t->item) {
    bst_insert(i, &(t->left));
  } else
    bst_insert(i, &(t->right));
  }
}
```

We can also write `bst_insert` *iteratively*:

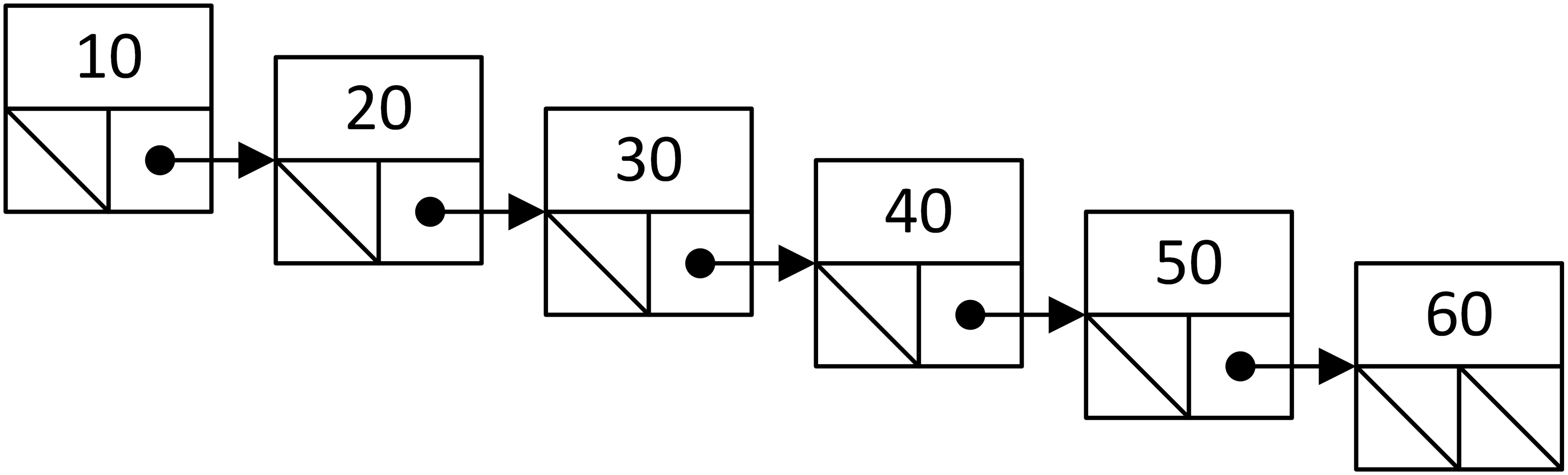
```
void bst_insert(int i, struct bstnode **ptr_root) {
  struct bstnode **ptr_t = ptr_root;
  while (*ptr_t != NULL) {
    struct bstnode *t = *ptr_t;
    if (t->item == i) {
      return;
    } else if (i < t->item) {
      ptr_t = &(t->left);
    } else {
      ptr_t = &(t->right);
    }
  }
  *ptr_t = make_bstnode(i, NULL, NULL);
}
```

Most tree functions are more easily written and understood with a recursive approach.

Trees and efficiency

What is the efficiency of `bst_insert`?

The *worst case* is when the tree is ***unbalanced***, and *every* node in the tree must be visited.



In this example, the run-time of `bst_insert` is $O(n)$, where n is the number of nodes in the tree.

In general, the running time of `bst_insert` depends more on the *height* of the tree than the *size* of the tree.

More accurately, the efficiency of `bst_insert` is $O(h)$, where h is the *height* of the tree.

The definition of a **balanced tree**, is a tree with a height of $O(\log n)$.

Conversely, the definition of an **un**balanced tree, is a tree with a height that is **not** $O(\log n)$.

Using the `bst_insert` function we provided, inserting the nodes in sorted order will clearly create an *unbalanced* tree.

With a **balanced** tree, `insert`, `remove` and `search` functions are all $O(\log n)$. With an **unbalanced** tree, each of these are $O(n)$.

In later CS courses, you will be introduced to ***self-balancing trees***.

An important feature of a *self-balancing tree* is to ensure that the `insert` and `remove` functions *preserve the balance of the tree* **and** remain $O(\log n)$.

Augmented approach

With linked lists, we were able to use a wrapper approach to store additional information in a wrapper structure to improve efficiency. For example, by storing the length we can have a $O(1)$ `length` function. A similar strategy can also be used with trees.

With trees, a new **augmented approach** is also popular.

With this approach, each *node* is *augmented* to include additional information about the node (or its subtree).

Self-balancing trees often use an augmented approach.

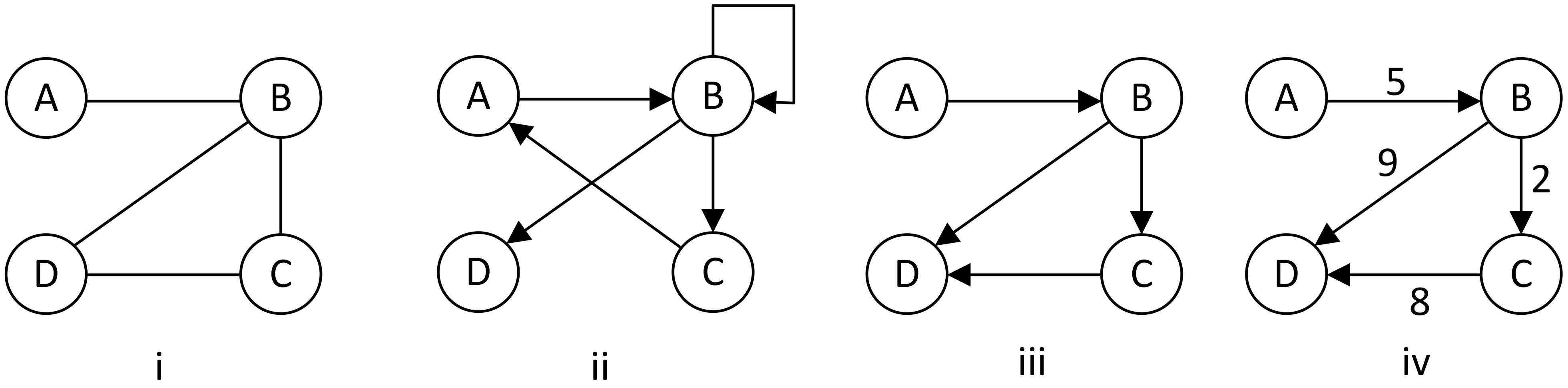
A popular augmentation is to store in *each node* the **size** of its subtree.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
  int size;                     // *****NEW
};
```

This augmentation not only allows us to retrieve the size of the tree in $O(1)$, it also makes it easier to retrieve the k-*th* item in a tree in $O(h)$ time.

Graphs

On a final note, linked lists and trees can be thought of as *"special cases"* of a ***graph*** data structure. Graphs are the only core data structure we are **not** working with in this course.



Graphs link **nodes** with **edges**. Graphs may be undirected (i) or directed (ii), allow cycles (ii) or be acyclic (iii), and have labelled edges (iv) or unlabeled edges (iii).

Goals of this module

You should be comfortable with the linked list and tree terminology introduced.

You should understand and be able to use linked lists and trees with functional, imperative, wrapper and augmented approaches.

You should understand the memory management issues related to working with linked lists and trees.

You should understand how the balance of a tree can affect the efficiency of tree functions.

You should understand how a wrapper or augmentation can be used to improve efficiency.