The primary goal of this section is to be able to use linked lists and trees.
Linked lists

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

Each *node* contains an *item* and a *link* (pointer) to the next node in the list.

The link in the last node is the **NULL**-pointer, a sentinel value.
Linked lists

Linked lists are usually represented as a link (pointer) to the front.

Unlike arrays, linked list nodes are not arranged sequentially in memory. There is no fast and convenient way to “jump” to the $i$-th element. The list must be traversed from the front. Traversing a linked list is $O(n)$. 
Linked lists

A significant advantage of a linked list is that its length can easily change, and the length does not need to be known in advance.

The memory for each node is allocated dynamically (i.e., using dynamic memory).
Linked lists in C: Structure definitions

A liist points to the front node (which is NULL for an empty list). Each llnode stores an item and a link (pointer) to the next node (which is NULL for the last node).

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

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

llnode is a recursive data structure (it has a pointer to its own structure type). llist is not recursive.
Creating a linked list

```c
// ll_create() creates a new, empty list.
// effects: allocates heap memory; client must call ll_destroy
// time:    O(1)
struct llist *ll_create(void) {
    struct llist *lst = malloc(sizeof(struct llist));
    lst->front = NULL;
    return lst;
}

int main(void) {
    struct llist *lst = list_create();
    // ...
}
```
Creating a linked list node

// llnode_create(item) creates a new node that stores item.  
// time:  O(1)
static struct llnode *llnode_create(int item) {
    struct llnode *node = malloc(sizeof(struct llnode));
    node->item = item;
    node->next = NULL;
    return node;
}

// llnode_destroy(node) releases all resources used by node.  
// time:  O(1)
static struct llnode *llnode_destroy(struct llnode * node) {
    assert(node);
    free(node);
}
Adding to the front

```c
void ll_add_front(struct llist *lst, int item) {
    struct llnode *new_node = llnode_create(item); // <- here
    new_node->next = lst->front;
    lst->front = new_node;
}
```
Adding to the front

// ll_add_front(lst, item) adds item to the front of
// the linked list lst.
// effects: modifies lst
void ll_add_front(struct llist *lst, int item) {
    struct llnode *new_node = llnode_create(item);
    new_node->next = lst->front; // <-- here
    lst->front = new_node;
}
Adding to the front

// ll_add_front(lst, item) adds item to the front of // the linked list lst. // effects: modifies lst

void ll_add_front(struct llist *lst, int item) {
    struct llnode *new_node = llnode_create(item);
    new_node->next = lst->front;
    lst->front = new_node; // <- here
}
Adding to the front

We could also add “allocates memory” as a side effect, but since most linked list functions use dynamic memory it becomes redundant. In addition, it is not necessarily memory that the client needs to worry about freeing.Specifying that it modifies the list is sufficient.

```
<table>
<thead>
<tr>
<th>str</th>
<th>char *</th>
<th>0x60</th>
<th>0x90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x70</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x74</td>
<td>0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x84</td>
<td>0x70</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x90</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x94</td>
<td>0x80</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
my_list struct llist * 0xB0 0x60
```

[Diagram of linked list]
Traversing a list: Iterative approach

We can traverse a list iteratively or recursively.

When iterating through a list, we typically use a (`llnode`) pointer to keep track of the “current” node.

```c
int ll_length(const struct llist *lst) {
    int len = 0;
    const struct llnode *current = lst->front;
    while (current) { // current != NULL
        ++len;
        current = current->next;
    }
    return len;
}
```

Remember (`current`) will be `NULL` (*false*) at the end of the list.
Traversing a list: Iterative approach

```c
int ll_length(const struct llist *lst) {
    int len = 0;
    const struct llnode *current = lst->front;
    while (current) {
        ++len; // <- here (2nd iteration)
        current = current->next;
    }
    return len;
}
```

```
<table>
<thead>
<tr>
<th>str</th>
<th>char *</th>
<th>0x60</th>
<th>0x90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x70</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x74</td>
<td>0x00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x84</td>
<td>0x70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x90</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x94</td>
<td>0x80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>my_list</th>
<th>struct llist *</th>
<th>0xB0</th>
<th>0x60</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>struct llnode *</td>
<td>0xB8</td>
<td>0x80</td>
</tr>
<tr>
<td>len</td>
<td>int</td>
<td>0xC0</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Traversing a list: Recursive approach

When using recursion, remember to recurse on a node (*llnode*) not the wrapper list itself (*lлист*).

```c
static int ll_length_wrk(struct llnode *node, int so_far) {
    if (node == NULL) {
        return so_far;
    } else {
        return ll_length_wrk(node->next, so_far + 1);
    }
}
```

You can write a corresponding wrapper function:

```c
int ll_length(struct lлист *lst) {
    return ll_length_wrk (lst->front, 0);
}
```
Destroying a list: Iterative approach

In C, we do not have a garbage collector, so we must be able to free our linked list. We need to free every node and the list wrapper.

When using an iterative approach, we are going to need two node pointers to ensure that the nodes are freed in a safe way.

```c
void ll_destroy(struct llist *lst) {
    struct llnode *cur_node = NULL;
    while (cur_node) {  // cur_node != NULL
        cur_node = lst->front
        lst->front = lst->front->next;
        llnode_destroy(cur_node);
    }
    free(lst);
}
```
void ll_destroy(struct llist *lst) {
    struct llnode *cur_node = NULL;
    while (cur_node) { // cur_node != NULL
        cur_node = lst->front // <- here
        lst->front = lst->front->next;
        llnode_destroy(cur_node);
    }
    free(lst);
}
Destroying a list: Iterative approach

```c
void ll_destroy(struct llist *lst) {
    struct llnode *cur_node = NULL;
    while (cur_node) { // cur_node != NULL
        cur_node = lst->front
        lst->front = lst->front->next; // <- here
        llnode_destroy(cur_node);
    }
    free(lst);
}
```
Destroying a list: Iterative approach

```c
void ll_destroy(struct llist *lst) {
    struct llnode *cur_node = NULL;
    while (cur_node) { // cur_node != NULL
        cur_node = lst->front
        lst->front = lst->front->next;
        llnode_destroy(cur_node); // <- here
    }
    free(lst);
}
```
Destroying a list: Recursive approach

With a recursive approach, it is more convenient to free the rest of the list before we free the first node.

```c
static void free_nodes(struct llnode *node) {
    if (node) {
        free_nodes(node->next);
        llnode_destroy(node);
    }
}

void ll_destroy(struct llist *lst) {
    free_nodes(lst->front);
    free(lst);
}
```
List duplication: Iterative approach

```c
struct llist *ll_duplicate(const struct llist *old_list) {
    struct llist *new_list = list_create();
    if (old_list->front) {
        add_front(old_list->front->item, new_list);
        const struct llnode *old_node = old_list->front->next;
        struct llnode *new_node = new_list->front;
        while (old_node) {
            new_node->next = llnode_create(old_node->item);
            new_node = new_node->next;
            old_node = old_node->next;
        }
    }
    return new_list;
}
```
List duplication: Recursive approach

```c
struct llnode *duplicate_nodes(const struct llnode *old_node) {
    if (old_node == NULL) {
        return NULL;
    } else {
        struct llnode *new_node = llnode_create(old_node->item);
        new_node->next = duplicate_nodes(old_node->next);
        return new_node;
    }
}

struct llist *ll_duplicate(const struct llist *old_list) {
    struct llist *new_list = list_create();
    new_list->front = duplicate_nodes(old_list->front);
    return new_list;
}
```
List insertion

When inserting into the “middle”, we need to find the node that will be before the new node we are inserting.

```
1 -> 3 -> 5 -> 7
```
List insertion
Imperative insert

// slst_insert(slst, item) inserts item into sorted list slst.
// effects: modifies slst
// time: O(n), where n is the length of slst

void slst_insert(struct llist *slst, int item) {
  if (slst->front == NULL || item < slst->front->item) {
    add_front(slst, item);
  } else {
    struct llnode *before = slst->front;
    while (before->next && item > before->next->item) {
      before = before->next;
    }
    struct llnode *new_node = llnode_create(item);
    new_node->next = before->next;
    before->next = new_node;
  }
}
Imperative insert

// slst_insert(slst, item) inserts item into sorted
// list slst.
// effects: modifies slst
// time: O(n), where n is the length of slst
void slst_insert(struct llist *slst, -7) {
    if (slst->front == NULL || -7 < slst->front->item) {
        add_front(slst, -7);
    } else {
        // ...
    }
}
Imperative insert

// slst_insert(slst, item) inserts item into sorted list slst.
// effects: modifies slst
// time: O(n), where n is the length of slst
void slst_insert(struct llist *slst, 4) {
    // ...
} else {
    struct llnode *before = slst->front;
    while (before->next != NULL && 4 > before->next->item) {
        before = before->next;
    } // <- here
    struct llnode *node_node = llnode_create(4);
    new_node->next = before->next;
    before->next = new_node;
}
Imperative insert

```c
// slst_insert(slst, item) inserts item into sorted list slst.
// effects: modifies slst
// time: O(n), where n is the length of slst
void slst_insert(struct llist *slst, 4) {
    // ...
    } else {
    struct llnode *before = slst->front;
    while (before->next != NULL && 4 > before->next->item) {
        before = before->next;
    }
    struct llnode *node_node = llnode_create(4); // <- here
    new_node->next = before->next;
    before->next = new_node;
}
```
Imperative insert

// slst_insert(slst, item) inserts item into sorted list slst.
// effects: modifies slst
// time: O(n), where n is the length of slst
void slst_insert(struct llist *slst, 4) {
    // ...
} else {
    struct llnode *before = slst->front;
    while (before->next != NULL && 4 > before->next->item) {
        before = before->next;
    }
    struct llnode *node_node = llnode_create(4);
    new_node->next = before->next; // <- here
    before->next = new_node;
}
Imperative insert

// slst_insert(slst, item) inserts item into sorted list slst.
// effects: modifies slst
// time: O(n), where n is the length of slst
void slst_insert(struct llist *slst, 4) {
    // ...
} else {
    struct llnode *before = slst->front;
    while (before->next != NULL && 4 > before->next->item) {
        before = before->next;
    }
    struct llnode *node_node = llnode_create(4);
    new_node->next = before->next;
    before->next = new_node; // <- here
}
Removing nodes

In C, we can implement a function that removes the first node. When removing nodes, make sure you do not create a memory leak.

```c
void ll_remove_front(struct llist *lst) {
    assert(lst->front);
    struct llnode *remove = lst->front;
    lst->front = lst->front->next;
    llnode_destroy(remove);
}
```
Removing nodes

```c
void ll_remove_front(struct llist *lst) {
    assert(lst->front);
    struct llnode *remove = lst->front; // <- HERE
    lst->front = lst->front->next;
    llnode_destroy(remove);
}
```
Removing nodes

```c
void ll_remove_front(struct llist *lst) {
    assert(lst->front);
    struct llnode *remove = lst->front;
    lst->front = lst->front->next; // <-- HERE
    llnode_destroy(remove);
}
```
Removing nodes

```c
void ll_remove_front(struct llist *lst) {
    assert(lst->front);
    struct llnode *remove = lst->front;
    lst->front = lst->front->next;
    llnode_destroy(remove); // <-- HERE
}
```
Removing nodes

Removing a node from an arbitrary list position is more complicated.

```c
// remove_item(lst, item) removes the first occurrence of item in
// lst and returns true if item is successfully removed, and
// false otherwise.
bool ll_remove_item(struct llist *lst, int item) {  
  if (lst->front == NULL) return false;
  if (lst->front->item == item) {
    remove_front(lst);
    return true;
  }
  struct llnode *before = lst->front;
  while (before->next && item != before->next->item) {
    before = before->next;
  }
  if (before->next == NULL) return false;
  struct llnode *remove = before->next;
  before->next = before->next->next;
  llnode_destroy(remove);
  return true;
}
```
Removing nodes

```c
bool ll_remove_item(struct llist *lst, 5) {
    // ...
    struct llnode *before = lst->front;
    while (before->next && 5 != before->next->item) {
        before = before->next;
    } // <- HERE
    if (before->next == NULL) return false;
    struct llnode *remove = before->next;
    before->next = before->next->next->next;
    llnode_destroy(remove);
    return true;
}
```
Removing nodes

```c
bool ll_remove_item(struct llist *lst, 5) {
    // ...
    struct llnode *before = lst->front;
    while (before->next && 5 != before->next->item) {
        before = before->next;
    }
    if (before->next == NULL) return false;
    struct llnode *remove = before->next; // <- HERE
    before->next = before->next->next->next;
    llnode_destroy(remove);
    return true;
}
```
Removing nodes

```c
bool ll_remove_item(struct llist *lst, 5) {
    // ...
    struct llnode *before = lst->front;
    while (before->next && 5 != before->next->item) {
        before = before->next;
    }
    if (before->next == NULL) return false;
    struct llnode *remove = before->next;
    before->next = before->next->next; // <-- HERE
    llnode_destroy(remove);
    return true;
}
```
Removing nodes

```c
bool ll_remove_item(struct llist *lst, 5) {
    // ...
    struct llnode *before = lst->front;
    while (before->next && 5 != before->next->item) {
        before = before->next;
    }
    if (before->next == NULL) return false;
    struct llnode *remove = before->next;
    before->next = before->next->next;
    llnode_destroy(remove); // <- HERE
    return true;
}
```
Caching information

Consider that we are writing an application where the length of a linked list will be queried often.

Typically, finding the length of a linked list is $O(n)$.

However, we can store (or “cache”) the length in the llist structure, so the length can be retrieved in $O(1)$ time.

```c
struct llist {
    struct llnode *front;
    int length;
};
```
Caching information

Naturally, other list functions would have to update the length as necessary:

• `ll_create` would initialize length to zero
• `ll_add_front`, `sllst_insert`, etc. would increment length
• `ll_remove_front`, `ll_remove_item`, etc. would decrement length
• ...


Data integrity

The introduction of the length field to the linked list may seem like a great idea to improve efficiency.

However, it introduces new ways that the structure can be corrupted. What if the length field does not accurately reflect the true length? For example, imagine that someone implements the ll_remove_item function, but forgets to update the length field?

Or a naïve coder may think that the following statement removes all of the nodes from the list:

```c
lst->length= 0;
```
Data integrity

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

Advanced testing methods can often find these types of errors, but you must exercise caution.

If data integrity is an issue, it is often better to repackage the data structure as a separate ADT module and only provide interface functions to the client.

This is an example of security (protecting the client from themselves).
Queue ADT

A queue is like a “lineup”, where new items go to the “back” of the line, and the items are removed from the “front” of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- add_back: adds an item to the end of the queue
- remove_front: removes the item at the front of the queue
- front: returns the item at the front
- is_empty: determines if the queue is empty
Queue ADT

A Stack ADT can be easily implemented using a dynamic array (as we did in Section 10) or with a linked list. While it is possible to implement a Queue ADT with a dynamic array, the implementation is a bit tricky. Queues are typically implemented with linked lists.

The only concern is that an add_back operation is normally $O(n)$. However, if we maintain a pointer to the back (last element) of the list, in addition to a pointer to the front of the list, we can implement add_back in $O(1)$.

Maintaining a back pointer is a popular modification to a traditional linked list, and another reason to use a wrapper.
Queue ADT: Interface

// all operations are O(1) (except destroy)
struct queue;
struct queue *queue_create(void);
void queue_destroy(struct queue *q);

void queue_enqueue(struct queue *q, int item);
int queue_dequeue(struct queue *q);
int queue_front(const struct queue *q);
bool queue_is_empty(const struct queue *q);
Queue ADT: Implementation

```c
struct queue {
    struct llnode *front;
    struct llnode *back; // new
};

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

struct queue *queue_create(void) {
    struct queue *q = malloc(sizeof(struct queue));
    q->front = NULL;
    q->back = NULL;
    return q;
}
```
Queue ADT: Implementation

```c
void queue_enqueue(struct queue *q, int item) {
    struct llnode *new = llnode_create(item);
    if (q->front == NULL) {
        q->front = new;
    } else {
        q->back->next = new;
    }
    q->back = new;
}
```
Queue ADT: Implementation

```c
int queue_dequeue(struct queue *q) {
    assert(q->front);
    int ret_val = q->front->item;
    struct llnode *remove = q->front;
    q->front = q->front->next;
    llnode_destroy(remove);
    if (q->front == NULL) {
        q->back = NULL;
    }
    return ret_val;
}
```
Queue ADT: Implementation

The remainder of the Queue ADT is straightforward.

```c
int queue_front(const struct queue *q) {
    assert(q->front);
    return q->front->item;
}

bool queue_is_empty(const struct queue *q) {
    return q->front == NULL;
}

void queue_destroy(struct queue *q) {
    while (!queue_is_empty(q)) {
        queue_remove_front(q);
    }
    free(q);
}
```
Node augmentation strategy

In a node augmentation strategy, each node is augmented to include additional information about the node or the structure.

For example, a dictionary node can contain both a key (item) and a corresponding value.

Or for a priority queue, each node can additionally store the priority of the item.
Node augmentation strategy

The most common node augmentation for a linked list is to create a doubly linked list, where each node also contains a pointer to the previous node. When combined with a back pointer in a wrapper, a doubly linked list can add or remove from the front and back in $O(1)$ time.

Many programming environments provide a double-ended Queue (dequeue or deque) ADT, which can be used as a Stack or a Queue ADT.
Trees

At the implementation level, trees are very similar to linked lists. Each node can link to more than one node.
Tree terminology

• the root node has no parent, all others have exactly one
• nodes can have multiple children
• in a binary tree, each node has at most two children
• a leaf node has no children
• the height of a tree is the maximum possible number of nodes from the root to a leaf (inclusive) (example: 3)
• the height of an empty tree is zero
• the number of nodes is known as the node count (example: 6)
Binary Search Trees (BSTs)

Binary Search Tree (BSTs) enforce the ordering property: for every node with an item \( i \), all items in the left child subtree are less than \( i \), and all items in the right child subtree are greater than \( i \).
BST: Implementation

Our BST node (*bstnode*) is very similar to our linked list node definition.

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

struct bst {
    struct bstnode *root;
};
```

In CS 135, BSTs were used as dictionaries, with each node storing both a key and a value. Traditionally, a BST only stores a single item, and additional values can be added as node augmentations if required.
BST: Implementation

As with linked lists, we need a function to create a new BST.

```c
// bst_create() creates a new BST
// effects: allocates memory: call bst_destroy
struct bst *bst_create(void) {
    struct bst *tree = malloc(sizeof(struct bst));
    tree->root = NULL;
    return tree;
}
```
BST: Implementation

Before writing code to insert a new node, first we write a helper to create a new leaf node.

```c
struct bstnode *node_create(int item) {
    struct bstnode *leaf = malloc(sizeof(struct bstnode));
    leaf->item = item;
    leaf->left = NULL;
    leaf->right = NULL;
    return leaf;
}
```

As with lists, we can write tree functions recursively or iteratively.
BST: Implementation

For the recursive version, we will need a wrapper, and we can handle the special case that the tree is empty.

```c
void bst_insert(struct bst *tr, int item) {
   if (tr->root) {
      node_insert(tr->root, item);
   } else {
      tr->root = node_create(item);
   }
}
```
BST: Implementation (recursive)

For the core function, we recurse on nodes.

```c
void node_insert(struct bstnode *node, int item) {
    if (item < node->item) {
        if (node->left) { // node->left != NULL
            node_insert(node->left, item);
        } else {
            node->left = node_create(item);
        }
    } else if (item > node->item) {
        if (node->right) { // node->right != NULL
            node_insert(node->right, item);
        } else {
            node->right = node_create(item);
        }
    } // else do nothing, as item already exists
}
```
void node_insert(struct bstnode *node, int item) {
    struct node *insert_after = NULL;
    while (node != NULL) {
        insert_after = node;
        if (item < node->item) {
            node = node->left;
        } else if (item > node->item) {
            node = node->right;
        } else {
            return;
        }
    }
    if (item < insert_after->item) {
        insert_after->left = node_create(item);
    } else if (item > insert_after->item) {
        insert_after->right = node_create(item);
    }
}
BST: iteration vs. recursion

Use iteration when you have to find a single node in the BST:

```c
void func(struct node *node) {
    struct node *parent = NULL;
    while (node && !comp(node)) {
        parent = node;
        if (comp(node) < 0) {
            node = node->left;
        } else { // comp(node) > 0
            node = node->right;
        }
    }
    // do something to parent
}
```

Use recursive when you have to traverse the entire BST:

```c
void func(struct node *node) {
    if (node != NULL) {
        func(node->left);
        func(node->right);
        // do something to node
    }
}
```
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 running time of bst_insert is $O(n)$, where $n$ is the number of nodes in the tree.
Trees and efficiency

The running time of `bst_insert` is $O(h)$: it depends more on the height of the tree ($h$) than the number of nodes in the tree ($n$).

The definition of a **balanced tree** is a tree where the height ($h$) is $O(\log n)$.

Conversely, an **unbalanced tree** is a tree with a height that is not $O(\log n)$. The height of an unbalanced tree is $O(n)$.

Using the `bst_insert` function we provided, inserting the nodes in sorted order creates an unbalanced tree.
Trees and efficiency

With a **balanced** tree, the running time of standard tree functions (e.g., insert, remove, search) are all $O(\log n)$.

With an **unbalanced** tree, the running time of each function is $O(n)$.

A **self-balancing** tree “re-arranges” the nodes to ensure that tree is always balanced.

With a good self-balancing implementation, all standard tree functions preserve the balance of the tree and have an $O(\log n)$ running time.

In CS 240 and CS 341 you will see self-balancing trees. Self-balancing trees often use node augmentations to store extra information to aid the re-balancing.
Count node augmentation

A popular tree node augmentation is to store in each node the count (number of nodes) in its subtree.

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

This augmentation allows us to retrieve the number of nodes in the tree in $O(1)$ time.

It also allows us to implement a select function in $O(h)$ time. `select(k)` finds item with index $k$ in the tree.
Count node augmentation
Count node augmentation

The following code illustrates how to select item with index k in a BST with a count node augmentation.

```c
int select_node(struct bstnode *node, int k) {
    assert(node && (0 <= k && k < node->count));
    int left_count = 0;
    if (node->left) left_count = node->left->count;
    if (k < left_count) return select_node(k, node->left);
    if (k == left_count) return node->item;
    return select_node(k - left_count - 1, node->right);
}

int bst_select(struct bst *tr, int k) {
    return select_node(k, tr->root);
}
```

`select(tr, 0)` finds the smallest item in the tree.
Dictionary ADT (revisited)

The dictionary ADT (also called a map, associative array, key-value store or symbol table), is a collection of pairs of **keys** and **values**.

Each key is unique and has a corresponding value, but more than one key may have the same value.

Typical dictionary ADT operations:

- **lookup**: for a given key, retrieve the corresponding value or “not found”
- **insert**: adds a new key/value pair (or replaces the value of an existing key)
- **remove**: deletes a key and its value
Dictionary ADT (revisited)

In the following example, we implement a Dictionary ADT using a BST data structure.
As in CS 135, we use int keys and string values.

// dictionary.h
struct dictionary;
struct dictionary *dict_create(void);
void dict_insert(struct dictionary *dict, int key, const char *val);
const char *dict_lookup(struct dictionary *dict, int key);
void dict_remove(struct dictionary *dict, int key);
void dict_destroy(struct dictionary *dict);
Dictionary ADT (revisited)

Using the same bstnode structure, we augment each node by adding an additional value field.

```c
struct bstnode {
    int key;
    char *value;
    struct bstnode *left;
    struct bstnode *right;
};

struct dictionary {
    struct bstnode *root;
};

struct dictionary *dict_create(void) {
    struct dictionary *dict = malloc(sizeof(struct dictionary));
    dict->root = NULL;
    return dict;
}
```
Dictionary ADT (revisited)

Our dictionary ADT operations are implemented with an iterative approach. The exception is the destroy operation. If a tree function needs to visit all children, a recursive approach is usually the best.

```c
void node_destroy(struct bstnode *node) {
    if (node) { // node != NULL
        node_destroy(node->left);
        node_destroy(node->right);
        free(node->value);
        free(node);
    }
}

void dict_destroy(struct dictionary *dict) {
    node_destroy(dict->root);
    free(dict);
}
```
Dictionary ADT (revisited)

This implementation of the lookup operation returns NULL if unsuccessful.

```c
const char *dict_lookup(const struct dictionary *dict, int key) {
    const struct bstnode *node = dict->root;
    while (node) { // node != NULL
        if (key < node->key) {
            node = node->left;
        } else if (key > node->key) {
            node = node->right;
        } else if (node->key == key) {
            return node->value;
        }
    }
    return NULL; // key was not found in dict
}
```
Dictionary ADT (revisited)

When inserting an int–string key–value pair into the dictionary, we need to make a copy of the string (value) passed by the client.

```
struct bstnode *node_create(int key, const char *value) {
    struct bstnode *node = malloc(sizeof(struct bstnode));
    node->item = key;
    node->value = my_strdup(value); // make a copy
    node->left = node->right = NULL;
    return node;
}
```

If a client tries to insert a key that already exists, we replace the value with the new value (as seen on the following slide).
Dictionary ADT (revisited)

```c
void dict_insert(struct dictionary *dict,  
                int key, const char *value) {
    struct bstnode *node = dict->root;
    struct bstnode *parent = NULL;
    while (node && node->item != key) {  
        parent = node;  
        if (key < node->item) {  
            node = node->left;
        } else if (key > node->item) {  
            node = node->right;
        }
    }
    if (node != NULL) {  // key already exists at node
        free(node->value);
        node->value = my_strdup(value);
    } else if (parent == NULL) {  // empty tree
        dict->root = node_create(key, value);
    } else if (key < parent->item) {  
        parent->left = node_create(key, value);
    } else {  
        parent->right = node_create(key, value);
    }
}
```
Dictionary ADT (revisited)

There are several different ways of removing a node (the “target”) from a BST. Our strategy is as follows:

1. If the target is a leaf, we remove it.
2. If a child of the target is NULL, the other child will “replace” the target.
3. If the target has two children, we will replace the target with the next largest key (the “replacement” node). The replacement node is the smallest key in the target’s right subtree. This replacement is not too complicated because the replacement node is guaranteed to have a NULL left child (otherwise, it would not be the next largest key).
Dictionary ADT (revisited)

There are several different ways of removing a node (the “target”) from a BST. Our strategy is as follows:

1. If the target is a leaf, we remove it.

2. If a child of the target is NULL, the other child will “replace” the target.

3. If the target has two children, we will replace the target with the next largest key (the “replacement” node). The replacement node is the smallest key in the target’s right subtree. This replacement is not too complicated because the replacement node is guaranteed to have a NULL left child (otherwise, it would not be the next largest key).
Dictionary ADT (revisited)

```c
void dict_remove(struct dictionary *dict, int key) {
    struct bstnode *target = dict->root;
    struct bstnode *target_parent = NULL;
    // find the target (and its parent)
    while (target && target->item != key) {
        target_parent = target;
        if (key < target->item) {
            target = target->left;
        } else if (key > target->item) {
            target = target->right;
        }
    }
    if (target == NULL) {
        return; // key not found
    }
    // ...
```
Dictionary ADT (revisited)

If either of the target’s children is NULL, the replacement is the other child. This also covers the case if the target is a leaf (the replacement is NULL).

```c
// ...  
// find the node to "replace" the target
struct bstnode *replacement = NULL;
if (target->left == NULL) {
    replacement = target->right;
} else if (target->right == NULL) {
    replacement = target->left;
} else { // neither child is NULL  
    // ...  
```
Dictionary ADT (revisited)

If the target has two children, the replacement is the next largest node (the smallest node in the target’s right subtree).

```c
// ...  
// find the replacement node and its parent
replacement = target->right;
struct bstnode *replacement_parent = target;
while (replacement->left) {
    replacement_parent = replacement;
    replacement = replacement->left;
}
// update the child links for the replacement and its parent
replacement->left = target->left;
if (replacement_parent != target) {
    replacement_parent->left = replacement->right;
    replacement->right = target->right;
}
// ...
```
Dictionary ADT (revisited)

Finally, the replacement has been found, and so the target must be freed and the parent of the target is updated.

```c
// ...
// free the target, and update the target's parent
free(target->value);
free(target);
if (target_parent == NULL) {
    dict->root = replacement;
} else if (key > target_parent->item) {
    target_parent->right = replacement;
} else {
    target_parent->left = replacement;
}
```
Array-based trees

For some types of trees, it is possible to use an array to store a tree.

• the root is stored at \(a[0]\)
• for the node at \(a[i]\)
  • its left is stored at \(a[2 * i + 1]\)
  • its right is stored at \(a[2 * i + 2]\)
  • its parent is stored at \(a[(i - 1) / 2]\)

• a special sentinel value can be used to indicate an empty node
• a tree of height \(h\) requires an array of length \(2h - 1\)
• (a dynamic array can be realloc’d as the tree height grows)
Array-based trees

parent: $i$
left: $2i + 1$
right: $2i + 2$
Array-based trees

Array-based trees are often used to implement “complete trees”, where there are no empty nodes, and every level of the tree is filled (except the bottom).

The heap data structure (not the section of memory) is often implemented as a complete tree in an array.

For self-balancing trees, the self-balancing (e.g., rotations) is often more awkward in the array notation. However, arrays work well with lazy rebalancing, where a rebalancing occurs infrequently (i.e., when a large imbalance is detected). The tree can be rebalanced in $O(n)$ time, typically achieving amortized $O(\log n)$ operations.
**Graphs**

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 labeled edges (iv) or unlabeled edges (iii).
Goals of this Section

At the end of this section, you should be able to:

• use the new linked list and tree terminology introduced
• use linked lists and trees with a recursive or iterative approach
• use wrapper structures and node augmentations to improve efficiency
• explain why an unbalanced tree can affect the efficiency of tree functions